SPRINGER NATURE
Reference

Anupam Chattopadhyay
Editor

Handbook of Computer Architecture

Springer

Handbook of Computer Architecture

Anupam Chattopadhyay
Editor

Handbook of Computer Architecture

With 672 Figures and 98 Tables

Springer

Editor
Anupam Chattopadhyay
College of Computing and Data Science
Nanyang Technological University
Singapore, Singapore

ISBN 978-981-97-9313-6 ISBN 978-981-97-9314-3 (eBook)
https://doi.org/10.1007/978-981-97-9314-3

© Springer Nature Singapore Pte Ltd. 2025

This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Singapore Pte Ltd.
The registered company address is: 152 Beach Road, #21-01/04 Gateway East, Singapore 189721, Singapore

If disposing of this product, please recycle the paper.

Preface

Computer architectures sit at an important juncture between ever-changing landscape of implementation technology and emerging applications. It presented with a crucial connect between the plane of abstract algorithm and detailed computing and storage technologies – through the lens of Instruction-Set Architectures (ISA) and micro-architecture. Over the years, the architectures evolved to cater to various applications, with the core objective of achieving better runtime, smaller area, and lesser power consumption. This led to pathbreaking innovations, which, in turn catapulted various application segments to prominence, such as wireless/mobile computing, graphics processing, and more recently machine learning. It is a difficult task to capture such momentum with various tributaries in a single volume and yet, this is what is attempted in the current *Handbook of Computer Architectures*. While the content follows a standard narrative, the format of this book is unique. It offers *live update* of the content, thereby allowing the authors to include modern developments within the same context of a specific topic.

The content is spread over multiple sections, and in each section, specific chapters offer a detailed glimpse of a topic of interest. The chapters are presented in increasing order of advanced concepts. It is also cross-linked in such a manner that reader can peruse a chapter with only necessary pre-requisite from selected, prior chapters.

In the first section of single-core processors, three chapters provide the background of computer organization, microarchitecture, and communication networks. This is complemented with chapters on operating systems, edge computing, and secure computing architectures – which provide sufficient foundation for a reader to move toward more advanced notions in any of the following sections.

The section on application-specific processors provides valuable insights into the growing demands from application developers to have customized architectures, also referred to as co-processors or accelerators. From a wide range of application segments, multimedia processing, scientific computing, machine learning, and cryptographic workloads are chosen to be covered here. Since these applications heavily depend on digital arithmetic, a short overview of the concepts is presented as well. Multimedia, machine learning, and several other domain-specific architectures are known to get influenced – for good or worse – due to the device-level faults appearing in advanced technology nodes. This is discussed in the section of fault-tolerant architectures.

Various application-specific processors and general-purpose ones come together to contribute in the rich tapestry of modern System-on-Chips (SoCs). This also enhances the notion of architectures significantly by offering reconfigurability as a property. Multicore SoCs and reconfigurable architectures are studied in a dedicated section, covering general-purpose multicore architectures, Graphics Processing Units (GPUs), and Field Programmable Gate Arrays (FPGAs). Furthermore, readers are offered to delve into the Coarse-Grained Reconfigurable Architectures (CGRAs), dynamic and partial reconfigurability notions as well as power management challenges for multicore systems.

Growing technology prowess offers various capabilities to modern architects. In the section of Emerging Computing Architectures, these are studied, including compute-in-memory architectures, architectures for microfluidic biochips, Quantum computing, and the ones benefitting from 3D ICs.

The complexity of modern computer architectures can only be managed with the help of powerful design automation flows. This is discussed in the section on Processor Design and Programming Flows. The introductory chapters on parallel programming models and dataflow models help reader to familiarize with the abstract notions necessary to grasp the design automation concepts. This foundation brings further the methodologies for design space exploration, followed by specific tool-flows, as elaborated in the chapters on architecture description languages, high-level synthesis, processor simulation, and virtual prototyping. For customizable, application-specific, and reconfigurable architectures, the compilation flows present a critical role to extract maximum efficiency out of the computing fabric. These are discussed in two chapters on FPGA-specific compilers and retargetable compilers. Balancing of technology constraints all the way to the application layer is a complex design automation challenge, which is discussed in the chapter on approximate computing architectures.

The last section of this volume brings forth the classic and modern techniques for testing and verification of computer architectures. After establishing the basic concepts of verification and testing, readers are invited to study techniques like model checking, formal equivalence, theorem proving, and concolic testing. Furthermore, information flow verification and symbolic simulation are discussed in detail, which find relevance in security assurance and data path circuits, respectively. Lastly, the verification of quantum circuits is also presented.

Overall, this volume encompasses nearly all major trends in computer architectures, including the design automation flows, application demands, and technology trends. We welcome readers to delve deep into the content with the ardent hope of benefitting the community with a useful, comprehensive, up-to-date reference of this topic.

Singapore, Singapore Anupam Chattopadhyay
December 2024

Acknowledgments

This book presents a collective effort of several years. It would be impossible to list everyone, who contributed, directly or indirectly, to the production of this volume. In the following, a partial list of contributions and acknowledgment is offered.

I would first like to thank Springer representatives, Ramesh Premnath and Stephen Yeung, who initially pitched this idea and helped to kickstart the book concept. Avi Mendelson, in the early ideation stage, presented critical feedback on the content and organization. I cannot but, express my most sincere gratitude to the section editors, not listed in any particular order, Suhaib Fahmy, Mohamed M. Sabry Aly, Jeronimo Castrillon, Grant Edmund Martin, and Sayak Ray. They not only helped shape the book by presenting ideas on the content distribution but also kept in close correspondence with the chapter authors for timely chapter submission and reviewing the iterations, thereby ensuring the high quality of the volume. Considering the fact that several section editors themselves contributed chapters in this book, the effort is truly grand, for which I remain very thankful.

The chapter authors (nearly 100!) spent considerable time to summarize the vast content of the computer architecture topics - relevant to their expertise - within constraints of space. On behalf of section editors, I am sincerely thankful to the authors for their valuable contribution.

Last but, not the least – Salmanul Faris Nedum Palli and Daniel Diwakar, who were production editors of this book at various stages, worked tirelessly throughout the production. I remain really thankful to them for the tremendous effort over years.

Contents

Volume 1

Part I Single Core Processors 1

1 **Microarchitecture** 3
Freddy Gabbay

2 **The Architecture** 47
Avi Mendelson

3 **Architectures for Self-Powered Edge Intelligence** 89
Amit Ranjan Trivedi, Jaeha Kung, and Jong Hwan Ko

4 **Real-Time Scheduling for Computing Architectures** 127
Arvind Easwaran, Michael Yuhas, Saravanan Ramanathan, and Ankita Samaddar

5 **Secure Processor Architectures** 171
Nikhilesh Singh, Vinod Ganesan, and Chester Rebeiro

6 **Bus and Memory Architectures** 201
Trevor E. Carlson

Part II Application-Specific Processors 213

7 **Architectures for Multimedia Processing: A Cross-Layer Perspective** 215
Muhammad Shafique and Bharath Srinivas Prabakaran

8 **Post-Quantum Cryptographic Accelerators** 237
Ayesha Khalid and Dur-e-Shahwar Kundi

9 **Fault Tolerant Architectures** 277
Siva Satyendra Sahoo, Anup Das, and Akash Kumar

10 **Architectures for Machine Learning** 321
Yongkui Yang, Chao Chen, and Zheng Wang

11 Computer Arithmetic 381
Farhad Merchant

12 Architectures for Scientific Computing 401
Farhad Merchant

Part III Multicore and Reconfigurable Architectures **415**

13 Field-Programmable Gate Array Architecture 417
Andrew Boutros and Vaughn Betz

14 Coarse-Grained Reconfigurable Array (CGRA) 465
Zhaoying Li, Dhananjaya Wijerathne, and Tulika Mitra

15 Dynamic and Partial Reconfiguration of FPGAs 507
Suhaib A. Fahmy and Krishnan B. Iyer

16 GPU Architecture 531
Hyeran Jeon

17 Power Management of Multicore Systems 561
Behnaz Ranjbar, Amit Kumar Singh, Siva Satyendra Sahoo, Piotr Dziurzanski, and Akash Kumar

18 General-Purpose Multicore Architectures 595
Saugata Ghose

Volume 2

Part IV Emerging Computing Architectures **645**

19 Compute-in-Memory Architecture 647
Hongwu Jiang, Shanshi Huang, and Shimeng Yu

20 Design Automation Techniques for Microfluidic Biochips 687
Xing Huang, Tung-Che Liang, Zhanwei Zhong, Tsung-Yi Ho, and Krishnendu Chakrabarty

21 Architectures for Quantum Information Processing 723
Suryansh Upadhyay, Mahabubul Alam, and Swaroop Ghosh

22 Design and Tool Solutions for Monolithic Three-Dimensional Integrated Circuits 751
Kyungwook Chang and Sung Kyu Lim

Part V Processor Design and Programming Flows **805**

23 Architecture Description Languages 807
Anupam Chattopadhyay, Zheng Wang, and Grant Edmund Martin

24 Accelerator Design with High-Level Synthesis 841
Christian Pilato and Stephanie Soldavini

25 Processor Simulation and Characterization 875
Grant Edmund Martin, Suhas Madhusudana, Greg Efland, and Vadim Kustov

26 Methodologies for Design Space Exploration 915
Andy D. Pimentel

27 Virtual Prototyping of Processor-Based Platforms 947
Tim Kogel

28 FPGA-Specific Compilers 989
Nitish Srivastava, Gai Liu, Yi-Hsiang Lai, and Zhiru Zhang

29 Approximate Computing Architectures 1027
Muhammad Abdullah Hanif, Vojtech Mrazek, and Muhammad Shafique

30 Parallel Programming Models 1069
Muhammad Nufail Farooqi, Mustafa Abduljabbar, Vicenç Beltran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Miquel Pericàs

31 Dataflow Models of Computation for Programming Heterogeneous Multicores 1107
Jeronimo Castrillon, Karol Desnos, Andrés Goens, and Christian Menard

32 Retargetable Compilation 1147
Gert Goossens, Dirk Lanneer, Johan Van Praet, and Werner Geurts

Part VI Test and Verification **1189**

33 Verification and Its Role in Design of Modern Computers 1191
Sayak Ray

34 Bit-Level Model Checking 1203
Alexander Ivrii and Yakir Vizel

35 High-Level Formal Equivalence 1243
Theo Drane and M. V. Achutha Kiran Kumar

36 Verification of Arithmetic and Datapath Circuits with Symbolic Simulation 1269
Roope Kaivola and John O'Leary

37 Microprocessor Assurance and the Role of Theorem Proving 1321
Shilpi Goel and Sandip Ray

38 Versatile Binary-Level Concolic Testing 1365
Bo Chen and Fei Xie

39 Information Flow Verification 1389
Cynthia Sturton and Ryan Kastner

40 Verification of Quantum Circuits 1413
Robert Wille and Lukas Burgholzer

Index 1441

About the Editor

Anupam Chattopadhyay received his B.E. degree from Jadavpur University, India; MSc. from ALaRI, Switzerland; and PhD from RWTH Aachen in 2000, 2002, and 2008, respectively. From 2008 to 2009, he worked as a Member of Consulting Staff in CoWare R&D, Noida, India. From 2010 to 2014, he led the MPSoC Architectures Research Group in RWTH Aachen, Germany, as a Junior Professor. Since 2014, he is with the College of Computing and Data Science (CCDS), Nanyang Technological University (NTU), where he is currently appointed as an associate professor, and also holds a courtesy appointment at School of Physical and Mathematical Sciences (SPMS), NTU. In the past, he held visiting positions at Politecnico di Torino, Italy; École Polytechnique Fédérale de Lausanne (EPFL), Switzerland; Technion, Israel; Kyoto University, Japan; and Indian Statistical Institute, Kolkata. During his doctoral studies, he worked on automatic Register Transfer Level (RTL) generation from the architecture description language LISA, which led to a spin-off, and subsequently was acquired by a leading Electronic Design Automation (EDA) vendor. He developed novel high-level optimizations, verification techniques, and proposed a language-based modeling, exploration and design framework for partially re-configurable processors-many of which resulted in successful technology transfers to the EDA and Semiconductor IP industry.

Anupam currently heads a team of 20+ researchers, overseeing projects in the area of computer architectures, security, design automation, and emerging technologies. His research advances have been reported in more than 250 conference/journal papers

(ACM/IEEE/Springer), multiple research monographs and edited books (CRC, Springer), and open-access forums. Anupam's research in the area of emerging technologies has been covered by major news outlets across the world, including *Asian Scientist*, *Straits Times*, and *The Economist*. Anupam regularly serves in the Technical Program Committees (TPCs) of top conferences, reviews journal/conference articles, and presented multiple invited seminars/tutorials in prestigious venues. He is a series editor of Springer book series on Computer Architecture and Design Methodologies. He is a senior member of Association for Computing Machinery (ACM) and Institute of Electrical and Electronics Engineers (IEEE). Anupam received Borcher's plaque from RWTH Aachen, Germany, for outstanding doctoral dissertation in 2008, nomination for the best IP award in the ACM/IEEE DATE Conference 2016, and nomination for the best paper award in the International Conference on VLSI Design 2018 and 2020.

Section Editors

Jeronimo Castrillon
Chair for Compiler Construction
cfaed – Center for Advancing Electronics Dresden
SCADS.AI – Center for scalable data analytics and artificial intelligence Dresden/Leipzig
6G-life Hub – Digital transformation and sovereignty of future communication networks
Technische Universität Dresden
Dresden, Germany

Suhaib A. Fahmy
King Abdullah University of Science and Technology (KAUST)
Department of Computer, Electrical and Mathematical Sciences and Engineering
Thuwal, Saudi Arabia

Grant Edmund Martin
Pleasanton, CA, USA

Sayak Ray
Intel Corporation
Intel Product Assurance and Security (IPAS)
San Jose, CA, USA

Mohamed M. Sabry Aly
Nanyang Technological University
College of Computing and Data Science (CDDS)
Singapore, Singapore

Grant Edmund Martin has retired

Contributors

Mustafa Abduljabbar The Ohio State University, Columbus, USA

Mahabubul Alam Pennsylvania State University, University Park, PA, USA

Vicenç Beltran Barcelona Supercomputing Center, Barcelona, Spain

Vaughn Betz Department of Electrical and Computer Engineering (ECE), University of Toronto, Toronto, ON, Canada

Andrew Boutros Department of Electrical and Computer Engineering (ECE), University of Toronto, Toronto, ON, Canada

Lukas Burgholzer Institute for Integrated Circuits, Johannes Kepler University Linz, Linz, Austria

Trevor E. Carlson Department of Computer Science, National University of Singapore, Singapore, Singapore

Jeronimo Castrillon Chair for Compiler Construction, TU Dresden, Dresden, Germany

Krishnendu Chakrabarty School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, AZ, USA

Kyungwook Chang Suwon, South Korea

Anupam Chattopadhyay School of Computer Science and Engineering, Nanyang Technological University, Singapore, Singapore

Bo Chen Intel Corporation, Hillsboro, OR, USA

Chao Chen Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, Shenzhen, China

Anup Das Drexel University, Philadelphia, PA, USA

Karol Desnos Univ Rennes, INSA Rennes, CNRS, IETR-UMR6164, Rennes, France

Theo Drane Intel Corporation, Folsom, CA, USA

Piotr Dziurzanski West Pomeranian University of Technology, Szczecin, Poland

Arvind Easwaran Nanyang Technological University, Singapore, Singapore

Greg Efland Cadence Design Systems, Tensilica R&D, San Jose, CA, USA

Suhaib A. Fahmy King Abdullah University of Science and Technology (KAUST), Department of Computer, Electrical and Mathematical Sciences and Engineering, Thuwal, Saudi Arabia

Muhammad Nufail Farooqi Leibniz Supercomputing Centre of the Bavarian Academy of Sciences and Humanities, Munich, Germany

Roger Ferrer Barcelona Supercomputing Center, Barcelona, Spain

Freddy Gabbay The Institute of Electrical Engineering and Applied Physics, The Hebrew University of Jerusalem, Jerusalem, Israel

Vinod Ganesan Indian Institute of Technology Madras, Chennai, India

Werner Geurts Synopsys, Leuven, Belgium

Saugata Ghose University of Illinois Urbana-Champaign, Urbana, IL, USA

Swaroop Ghosh Pennsylvania State University, University Park, PA, USA

Shilpi Goel Intel Corporation, Austin, TX, USA

Andrés Goens School of Informatics, The University of Edinburgh, Edinburgh, UK

Gert Goossens Synopsys, Leuven, Belgium

Muhammad Abdullah Hanif Engineering Division, New York University Abu Dhabi, Abu Dhabi, United Arab Emirates

Tsung-Yi Ho Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, China

Shanshi Huang School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, USA

Xing Huang School of Computer Science, Northwestern Polytechnical University, Xi'an, China

Alexander Ivrii IBM, Haifa, Israel

Krishnan B. Iyer Computer Science, King Abdullah University of Science and Technology, Thuwal, Saudi Arabia

Hyeran Jeon University of California Merced, Merced, CA, USA

Hongwu Jiang School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, USA

Roope Kaivola Core and Client Development Group, Intel Corporation, Hillsboro, OR, USA

Ryan Kastner University of California San Diego, La Jolla, CA, USA

Ayesha Khalid Centre for Secure Information Technologies (CSIT), Queen's University Belfast, Belfast, UK

M. V. Achutha Kiran Kumar DEG, Intel Corporation, Bengaluru, India

Jong Hwan Ko Sungkyunkwan University (SKKU), Suwon, Republic of Korea

Tim Kogel Synopsys, Inc., Aachen, Germany

Akash Kumar Technische Universität Dresden, Dresden, Germany

Dur-e-Shahwar Kundi Centre for Secure Information Technologies (CSIT), Queen's University Belfast, Belfast, UK

Jaeha Kung Daegu Gyeongbuk Institute of Science and Technology (DGIST), Daegu, Republic of Korea

Vadim Kustov Cadence Design Systems, Tensilica R&D, San Jose, CA, USA

Yi-Hsiang Lai Cornell University, Ithaca, NY, USA

Dirk Lanneer Synopsys, Leuven, Belgium

Zhaoying Li National University of Singapore, Singapore, Singapore

Tung-Che Liang Department of Electrical and Computer Engineering, Duke University, Durham, NC, USA

Sung Kyu Lim Atlanta, USA

Gai Liu Xilinx, Inc., San Jose, CA, USA

Suhas Madhusudana Cadence Design Systems, Tensilica R&D, San Jose, CA, USA

Grant Edmund Martin Pleasanton, CA, USA

Xavier Martorell Barcelona Supercomputing Center, Barcelona, Spain

Christian Menard Chair for Compiler Construction, TU Dresden, Dresden, Germany

Avi Mendelson CS Department, Technion, Haifa, Israel

Farhad Merchant University of Groningen, Groningen, The Netherlands

Tulika Mitra National University of Singapore, Singapore, Singapore

Vojtech Mrazek Faculty of Information Technology, Brno University of Technology, Brno, Czech Republic

John O'Leary Core and Client Development Group, Intel Corporation, Hillsboro, OR, USA

Miquel Pericàs Chalmers University of Technology, Gothenburg, Sweden

Christian Pilato Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milano, Italy

Andy D. Pimentel Parallel Computing Systems Group, University of Amsterdam, Amsterdam, The Netherlands

Bharath Srinivas Prabakaran Institute of Computer Engineering, Technische Universität Wien (TU Wien), Vienna, Austria

Saravanan Ramanathan Nanyang Technological University, Singapore, Singapore

Behnaz Ranjbar Technische Universität Dresden, Dresden, Germany

Sandip Ray Department of ECE, University of Florida, Gainesville, FL, USA

Sayak Ray Intel Corporation, San Jose, CA, USA

Chester Rebeiro Indian Institute of Technology Madras, Chennai, India

Siva Satyendra Sahoo Technische Universität Dresden, Dresden, Germany

Ankita Samaddar Nanyang Technological University, Singapore, Singapore

Muhammad Shafique Engineering Division, New York University Abu Dhabi, Abu Dhabi, United Arab Emirates

Nikhilesh Singh Indian Institute of Technology Madras, Chennai, India

Amit Kumar Singh University of Essex, Colchester, UK

Stephanie Soldavini Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milano, Italy

Nitish Srivastava Google LLC, Mountain View, CA, USA

Cynthia Sturton University of North Carolina at Chapel Hill, Chapel Hill, NC, USA

Xavier Teruel Barcelona Supercomputing Center, Barcelona, Spain

Amit Ranjan Trivedi University of Illinois at Chicago, Chicago, IL, USA

Suryansh Upadhyay Pennsylvania State University, University Park, PA, USA

Johan Van Praet Synopsys, Leuven, Belgium

Yakir Vizel Technion - Israel Institute of Technology, Haifa, Israel

Zheng Wang Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, Shenzhen, China

Dhananjaya Wijerathne National University of Singapore, Singapore, Singapore

Robert Wille Chair for Design Automation, Technical University of Munich, Munich, Germany

Software Competence Center Hagenberg GmbH (SCCH), Hagenberg im Mühlkreis, Austria

Fei Xie Department of Computer Science, Portland State University, Portland, OR, USA

Yongkui Yang Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, Shenzhen, China

Shimeng Yu School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, USA

Michael Yuhas Nanyang Technological University, Singapore, Singapore

Zhiru Zhang Cornell University, Ithaca, NY, USA

Zhanwei Zhong Department of Electrical and Computer Engineering, Duke University, Durham, NC, USA

Part I
Single Core Processors

Microarchitecture

1

Freddy Gabbay

Contents

Introduction 4
Single-Cycle Processor Design 5
Processor Data Path 6
Processor Control Unit 11
Pipelining 12
Pipeline Principle and Performance Metrics 12
Pipelined Processors 16
Pipeline Hazards 18
Multiple-Issue Processor 35
Conclusions 43
References 45

Abstract

In the modern era of computing, microprocessors continue being at the heart of many computer systems. The different requirements by varieties of applications from edge devices, IoT, mobile, personal computers, and high-performance systems introduce major challenges on processor microarchitecture. These challenges have been further heightened when advance VLSI process technologies ceased from providing frequency scale to fuel processor performance growth. In modern systems, additional factors such as scale, power, and cost have become crucial design factors. As a result, all these challenges have intensified the importance of processor microarchitecture in conjunction with the physical constraints and application requirements. In this chapter, different processor microarchitectures are comprehensively covered. Through the discussion, the

F. Gabbay (✉)
The Institute of Electrical Engineering and Applied Physics, The Hebrew University of Jerusalem, Jerusalem, Israel

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_2

objective is to study design implementation details while examining hardware efficiency and performance. The chapter starts with a single-cycle processor and overview processor data-path and control unit. Next, the chapter introduces pipelined processors and presents the design complexity and performance aspects. Finally, multiple-issue processors such as superscalar, VLIW, and out-of-order processors are presented. The limitations and challenges of such processors to exploit the available parallelism of instructions are presented. The chapter is concluded by summarizing the future trends and challenges for processor microarchitectures.

Keywords

Processor microarchitecture · Single-cycle core · Pipeline · Superscalar · VLIW · Out-of-order · Instruction-level parallelism

Introduction

In the past decades, CPUs have been challenged by incredibly growing number of applications driven by the Internet revolution followed by the mobile and the data revolutions which affect every field in everyday life. The growing demand for performance, scale, and the diversity of use cases have continuously fueled the need for advanced processor microarchitecture which can satisfy these goals. For several decades, the advanced VLSI technology has been providing backwind to processor performance through continuous increase in frequency. Even though at the first decade of the twenty-first century, Moore's law for frequency has ceased to exist, the need for powerful processors has been continuously growing. Architects have been required to bring revolutionary microarchitectural innovations to satisfy the growing demand for powerful processing. Artificial intelligence applications and high-performance computing have even heightened the performance bar by introducing a demand for processing power in the scale of exaflops. Along with the need for high-performance processors, power considerations have become crucial factor not only for mobile applications and edge devices but also in cloud servers and datacenters where power is a major part of operating expenses. New applications such as IoT, wearable devices, edge computing, and the automotive market introduced specific needs for customized processor where performance is not always the ultimate goal, but rather cost, die area, real-time considerations, and heterogeneous integration of processors, peripherals, and accelerators are in the main interest.

In this chapter a processor microarchitecture is introduced stepwise by relying on digital building blocks. The chapter starts with a single-cycle processor microarchitecture where the design of both data-path and control unit is presented. The chapter broadly discusses microarchitectural design considerations of the single-cycle processors and presents the metrics for performance evaluation. Next, the pipelined processor core is presented with the data-path and control

unit implementation details. As part of the pipelined processor design, pipeline hazards are comprehensively discussed, and various microarchitectural solutions are presented. Finally, the multi-issue processor is introduced with a discussion on the concept of instruction-level parallelism (ILP) represented by the dataflow graph. As part of this discussion, the limitations of these processors to exploit the available ILP are discussed, which are governed by both control dependencies and data dependencies. In this last section, various multi-issue microarchitectures are presented: the superscalar, the VLIW, and the out-of-order processors. Through these discussions, microarchitectural building elements are introduced, and various related challenges, solutions, and trade-offs are debated.

Single-Cycle Processor Design

A common processor design relies on the Von Neumann machine model which was introduced by 1945 by John Von Neumann. The Von Neumann model proposes an architecture for a digital computer which consists of the following elements:

- A central processing unit (CPU) that contains arithmetic logic unit (ALU, also known as data path), local registers, and a control unit
- Memory unit that stores program instructions and data
- Input and output devices such as an external storage device, a network connection, a display, a keyboard, etc. (Fig. 1).

A similar processor model to the Von Neumann model is the Harvard architecture (Sloss et al. 2004) depicted by Fig. 2. In the Harvard architecture scheme, a separate memory is used for the program instructions and program data.

The Harvard architecture machine model will be the baseline for the processor design to be discussed further. In this chapter, the processor architecture is assumed to be a Reduced Instruction Set Architecture (RISC) (Hennessy and Patterson 2011) similar to MIPS (Hennessy and Patterson 2011) or RISCV (Patterson and Watterman 2017) processors that employ the following types of instructions (Table 1).

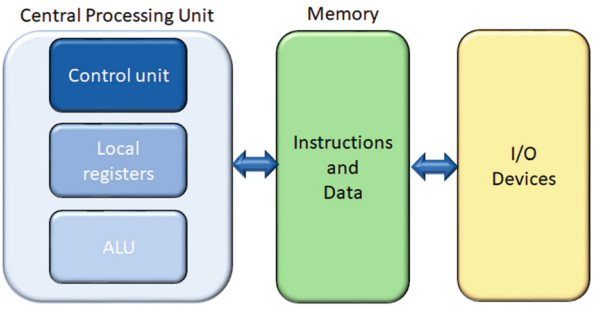


Fig. 1 Von Neumann machine model

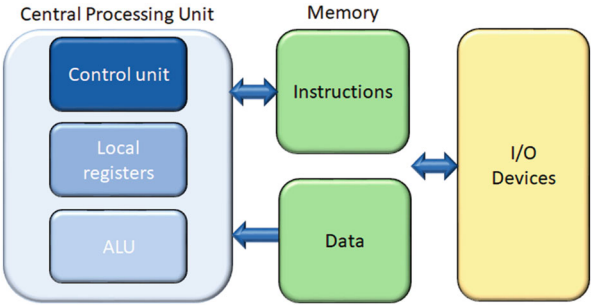


Fig. 2 The Harvard architecture

Table 1 Instruction types

Type	Instructions
ALU	Arithmetic logic instructions with register-register or register-immediate source operands, for example: add r1, r2, r3 andi r1, r2, 100

Load/Store	Memory access instructions. Load for read memory access and store for write memory access. Memory effective address (pointer) is determined by the displacement addressing mode, for example:
	lw r1, 100(r2)
	sw r1, 100(r2)

Control	Any instruction that can change the program countervalue, for example, Conditional direct branches: beq r1, r2, loop Unconditional direct jumps: jump loop Unconditional indirect jumps: jump r31

Processor Data Path

This subsection starts by introducing a single-cycle processor where every instruction is executed through a single machine cycle. An instruction execution is partitioned into the following data-path phases:

- Fetch – an instruction is fetched from memory, and the program counter (PC) is incremented to the next instruction.
- Decode – an instruction is decoded; the control unit generates the needed control signals for the data path. Source registers are read (when applicable) from the register file.

- Execution – ALU instructions are processed by the ALU, load/store instruction calculates their memory effective address, and for control instructions the branch condition and the target address are calculated.
- Memory – Applicable only for load/store instructions where data in memory is accessed.
- Write-back – Computed result is written back (when applicable) to the register file.

The instruction fetch circuit is illustrated by Fig. 3. The pointer to the instruction to be fetched from memory is maintained by the program counter (PC) register. The PC is incremented by 4 (assuming 4-byte instruction size) every cycle, unless control instruction changes the PC sequence. When the PC sequential order changes, the new target address is loaded by the control signal PC-CTRL into the PC.

The instruction decode circuit is illustrated by Fig. 4. The process of decoding an instruction involves extracting the opcode, source register 1 (sreg1), source register 2 (sreg2), destination register (dreg), and sign-extended immediate (sxtimm) fields from the instruction binary code. In addition, sreg1, sreg2, and dreg signals are sent to the register file to specify the identifiers of registers to be accessed. The register file scheme, illustrated by this figure, consists of a bank of the architectural registers which are accessible by one write port and two read ports. The number of read and write ports is determined by the maximum number of source and destination operands of an instruction. In the processor architecture presented in this chapter, it is assumed up to two source operands and one destination operand. The read ports are implemented by two multiplexors, each controlled by the corresponding sreg1 and sreg2 signals. The outputs of the multiplexors provide the value of the source operands being read, sregval1 and sregval2. The write port is implemented by a decoder which asserts the enable signal to the register that corresponds to the

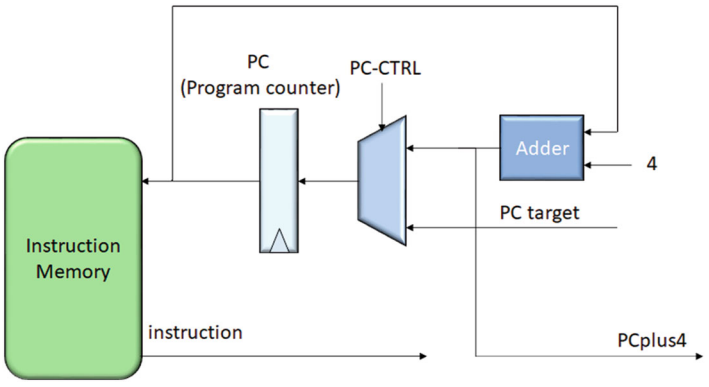


Fig. 3 Instruction fetch

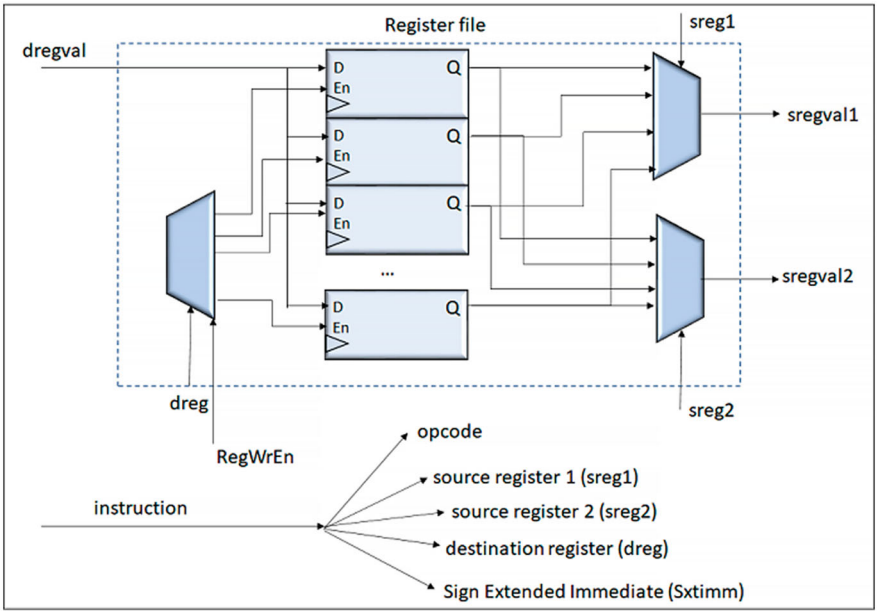


Fig. 4 Instruction decode

destination register, dreg. When the enable signal is asserted the write data, dregval is sampled by the corresponding register. Note that the decoder is controlled by the RegWrEn signal which enables writes to the register file. When RehWrEn=0 no write operations can be performed.

The execution circuit, illustrated by Fig. 5, performs the computation of results for ALU-type instructions. The sergval1 is connected to the first input port of the ALU, while the second port is connected to sregval2 or the sign-extended immediate value, sxtimm. The selection between the two possible options is performed by a multiplexor controlled by the selimm signal. For load/store instruction, the ALU calculates the effective address for the memory access instruction. The first port of the ALU is fed by the base register (through sregval1), while the displacement field is selected through the immediate field. For example, for lw r1, 100(r2) instruction, the base register r2 value, read from the register file, is sent to the ALU through the sregval1 signal, while the displacement, 100, is taken from the immediate field of the instruction, sign extended, and selected by the multiplexor for the second port of the ALU. Conditional control flow instructions are also processed by the illustrated circuit. Typically, such instructions are indirect branches which indicate their target address as an offset from the PC+4 in the immediate field of the instruction binary code. The sign-extended immediate field, sxtimm, is added to the PC+4 to calculate the target branch address in case that the control flow instruction

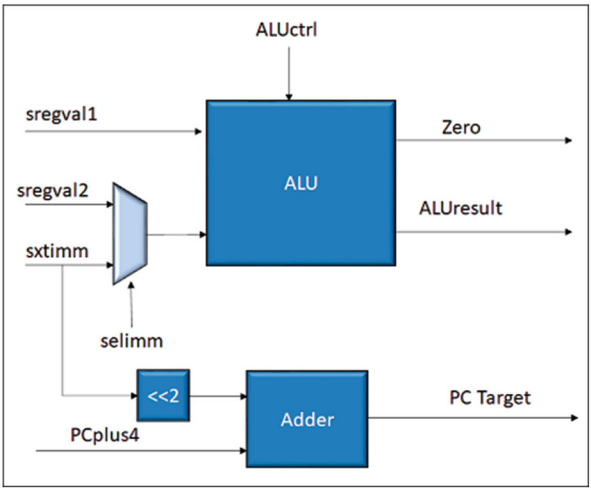


Fig. 5 Execution

is taken. In this case the new target address is loaded to the PC as it was illustrated by Fig. 3. Conditional control flow instructions are also required to evaluate the branch condition. In the presented design, it is assumed that there are two types of conditional branch instructions beq or bne (similar to MIPS architecture (Kane 1988)). For both instructions the two sources operand are compared by the ALU and in case they are equal, and the zero signal is set to 1; otherwise, the zero flag is set to 0. The zero signal is used in conjunction with the instruction opcode (beq or bne) by the control unit logic to generate the PC-ctrl signal depicted by Fig. 3.

The data memory access circuitry is illustrated by Fig. 6. The memory address, calculated by the ALU, is sent to the data memory address through the ALUresult signal. In case of data memory write (store instruction), the data to be written is read from the register file using the second source operand identifier. The register value, denoted by the sregval2, is connected to the Data in input of the data memory. In the case of memory read (load instruction), the data read from the data out port of the memory is connected to the Memout signal which is written to the register file by the write-back circuitry. The memory control signal MemWrEn and MemRdEn control the memory write and read operations, respectively. These signals are asserted by the control unit base on the instruction opcode.

Finally, the write-back circuitry is shown by Fig. 7. Write-back operations can be performed by either ALU instructions or load instructions. In accordance to the instruction opcode, the control unit sets the MemALUsel signal of the multiplexor to select between the ALUresult and the Memout values to be sent to the register file write port signal, dregval.

The full data path is illustrated by Fig. 8. The depicted data path is obtained by connecting all together the five circuits shown by the previous figures.

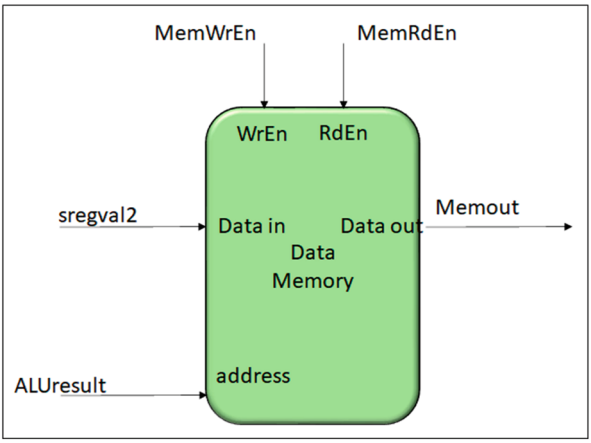


Fig. 6 Memory access

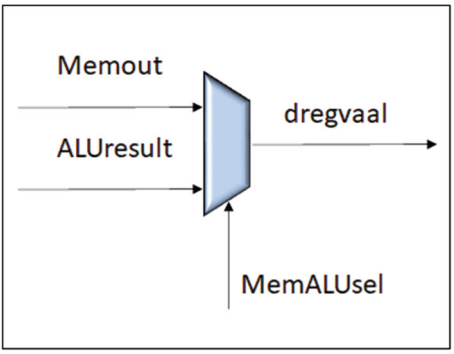


Fig. 7 Write-back

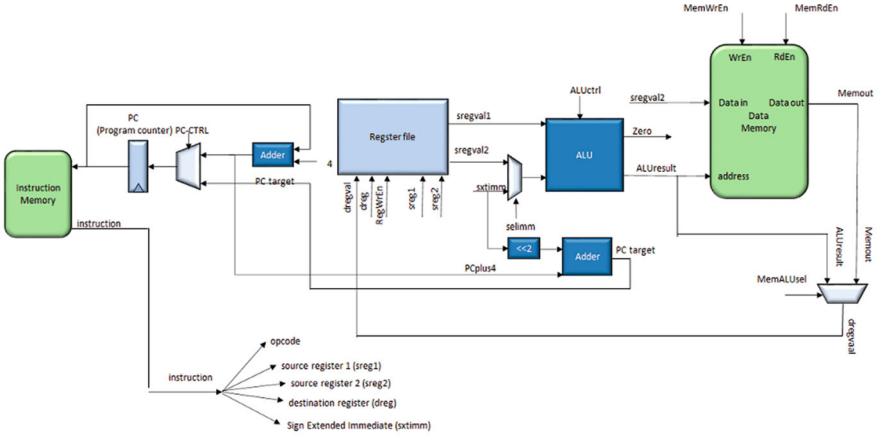


Fig. 8 Single-cycle processor data path

Processor Control Unit

The processor control unit provides the control signals to the core data path as shown by Fig. 9.

As it can be observed, the control unit has two input signals, the instruction opcode and the zero indication. The control unit output signals are PC-ctrl, ALUctrl, MemWrEn, MemRdEn, RegWrEn, and MemALUsel. The design of the control unit for a single-cycle core can be obtained as a combinatorial circuit which is described by the following truth tables for each of the instruction types (Tables 2, 3, and 4).

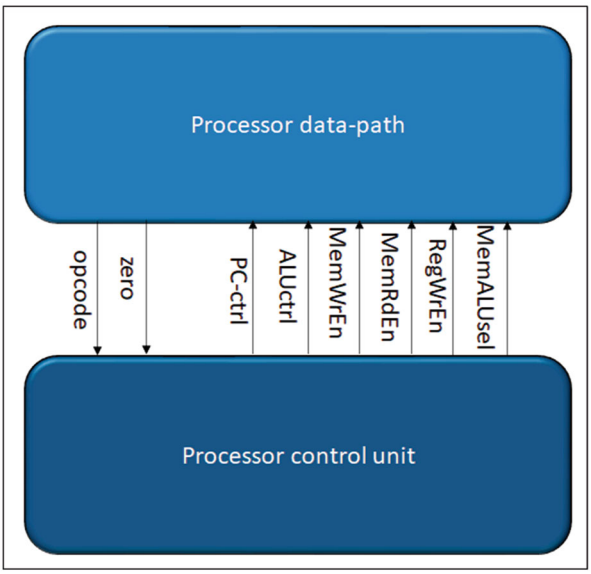


Fig. 9 Processor control unit and data path

Table 2 Control unit truth table for ALU instructions

Opcode	Zero	PC-ctrl	ALUctrl	MemWrEn	MemRdEn	RegWrEn	Mem ALUsel
ALU (non-immediate)	X	0	Determined by opcode	0	0	1	0
ALU (immediate)	X	0	Determined by opcode	0	0	1	0

Table 3 Control unit truth table for load/store instructions

Opcode	Zero	PC-ctrl	ALUctrl	MemWrEn	MemRdEn	RegWrEn	Mem ALUsel
Load	X	0	add	0	1	1	1
Store	X	0	add	1	0	0	x

Table 4 Control unit truth table for control flow instructions

Opcode	Zero	PC-ctrl	ALUctrl	MemWrEn	MemRdEn	RegWrEn	Mem ALUsel
beq	0	0	sub	0	0	0	0
	1	1	sub	0	0	0	0
bneq	0	1	sub	0	0	0	0
	1	0	sub	0	0	0	0
Jump	X	1	sub	0	0	0	0

Table 5 Instruction processing phases

Phase	ALU	Load/Store	Branch/Jump
Fetch	An instruction is fetched from the memory address pointed by the program pointer		
Decode	An instruction is decoded, source operands are read from the register file and data-path control signals are generated		
Execution	Arithmetic logic calculation is performed	Memory address is calculated	Target address is calculated, and branch condition is evaluated
Memory	n/a	Memory access for load or store	n/a
Write-back	Result is written to register file	Load value is written to register file	n/a

Pipelining

Now that the simple single-cycle processor has been designed, the next step is focused in examining its performance in quantitative measures. As a reminder, the single-cycle core, illustrated by Fig. 8, processes a single instruction every clock cycle. It was previously identified that the processing of an instruction includes the following phases summarized by Table 5 per instruction type.

As it can be observed by this table, the efficiency of the single-cycle process is relatively low. For example, once an instruction is fetched, the fetch hardware becomes idle through the rest of the phases till a new instruction is fetched again. Similar idleness can be identified for the other hardware mechanisms such as the decode logic, register file, ALU, and memory.

Pipeline Principle and Performance Metrics

Such underutilization of resources can be illustrated by comparing a single-cycle processor to a pizzeria, and the processing of instruction to the process of making a single pizza. The process of pizza making consists of the following sages: dough preparation, pizza topping (tomato, cheese, etc.), and baking. Suppose that every phase is staffed with one employee who is responsible to each of the corresponding phases. In addition, assuming that the duration of every stage is summarized as

Table 6 Processing stage of pizza making

	Phase	Duration	Utilization
	Dough preparation	8	8/20 = 40%
	Topping	2	2/20=10%
	Baking	10	10/20=50%
	Total time	20	

follows. As it can observed by the table below, the whole process of pizza making takes 20 min; however, the pizzeria staff utilization is far from being optimal. The dough preparation employee is utilized only 40% of the pizza preparation time, while the utilization of the topping employee is as small as 10%, and the baking employee is utilized 50% of the time. Such kind of pizza manpower is obviously inefficient due to the hidden unemployment at every stage which is illustrated by the following figure (Table 6).

The reason for the low utilization of the pizza line is due to the sequential processing of every task. Every pizza line employee does not start processing of a new pizza till the whole preparation of the previous pizza completed. In order to quantitatively examine the pizza line, throughput is defined as the number of jobs or tasks competed per time:

$$Throughput = \frac{Tasks\ Completed}{Time} \tag{1}$$

As it can be observed from Fig. 10, the throughput of the pizza line is one pizza per 20 min, or three pizzas per 1 h. The throughput and employee utilization can be significantly improved if the pizza processing is executed in a pipelined manner. Pipelining is a common technique which is used in varieties of applications such as manufacturing lines and many others. The concept of a pipeline relies on partitioning the processing of a job or a task into multiple pipeline stages, where every stage starts the process of a new task as soon as it is available. As a result, all pipeline stages work in parallel while processing different portion of different jobs respectively. Pipelining the pizza line is illustrated by Fig. 11.

As it can be observed from this figure, all the three pizza line workers work in parallel on three different pizzas. In addition, pizzas are moved from one stage to another every 10 min. This is determined by the slowest stage which is the pizza baking in the presented case. When examining the throughput of the pizza line, it is observed that a pizza is completed every 10 min or six pizzas per hour. Therefore, pipelining in this example gains a x2 throughput improvement with respect to the sequential pizza line. It can also be noticed that the preparation time of a single pizza has become even worse since it involves three pipeline stage of 10 min each resulting in a total preparation time of 30 min. In spite of the fact that the latency of pizza preparation gets worse, the employee utilization is significantly improved. For example, the dough preparation stage utilization is now 80%, while the topping stage utilization is 20% and the baking stage utilization is 100%.

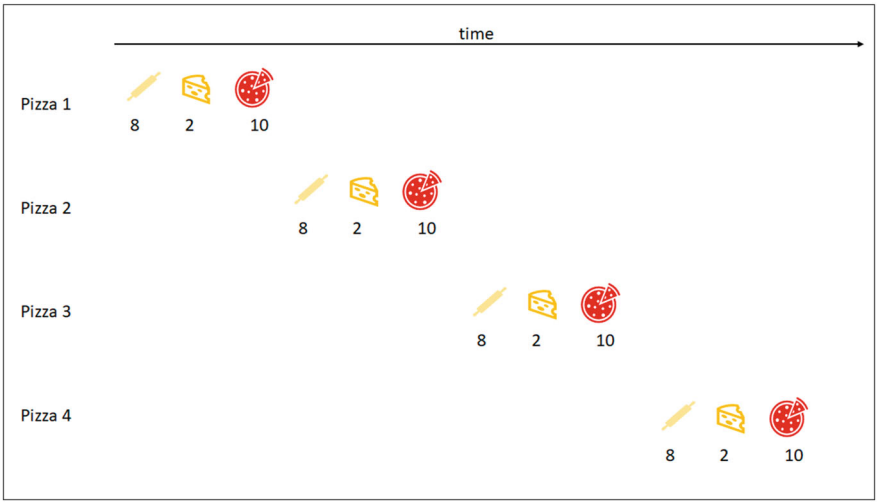
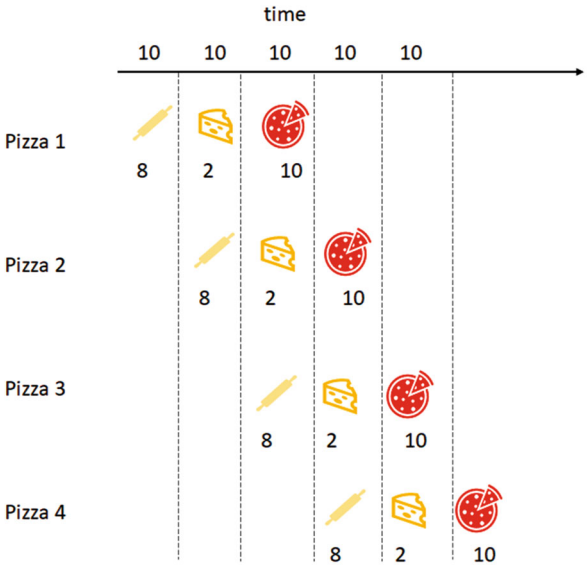


Fig. 10 Pizza processing stages

Fig. 11 Pipelined pizza line



All stages introduce significant improvement relative to the utilization numbers depicted earlier. Moving forward, the next question that is to be raised in this discussion is whether the utilization of the pizza line can be further improved. For example, the utilization of topping stage is improved to 20%, but is it possible to gain

further improvement? The answer is yes, but this of course depends on whether the tasks in every pipeline stage can be further divided into smaller stages. For example, if one can break both the dough preparation stage and the baking stage into two equal stages, then the following pizza line is obtained.

The pizza line illustrated by Fig. 12 consists of five pipeline stages. This time a pizza is moved from one stage to another every 5 min and therefore the throughput is six pizzas per 1 h. This yields a 2× throughput improvement with respect to the three-stage pizza line and 4× throughput improvement with respect to the sequential pizza line. Can one continue breaking the pipeline into more stages and gain an infinite throughput? Although it might be implied theoretically, practically pipeline stages cannot be infinitely broken into smaller stages, and this is due to several reasons. First, it is not always possible to break a given task into smaller tasks, since some operations may be indivisible (atomic) and second the process of moving the pizza from one stage to another involves a certain overhead which was not taken into account in the throughput calculation. For example, moving the pizza from one stage to another requires some extra time for each of the employees to carry the pizza to the next stage.

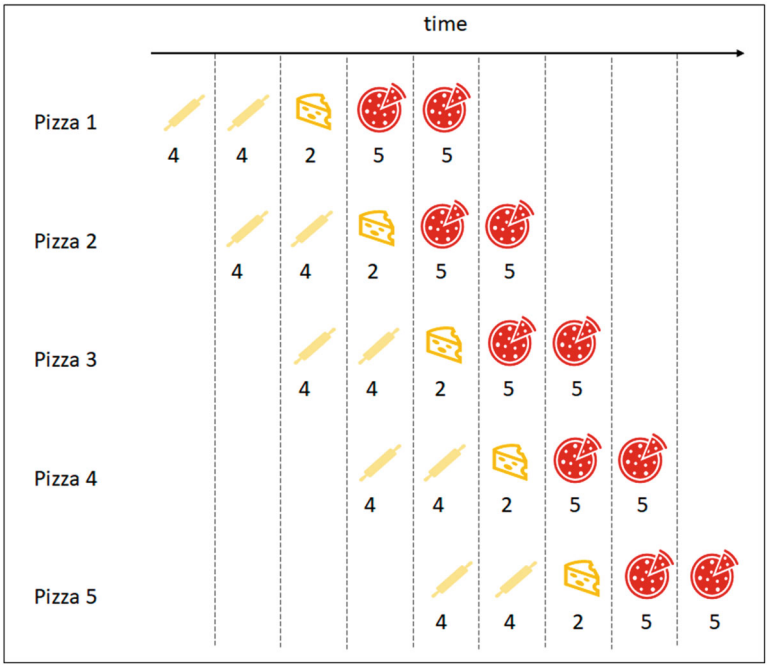


Fig. 12 Pipelined pizza line with five stages

Pipelined Processors

The single-cycle processor utilization and performance can be improved by employing the pipelining technique that was previously introduced and demonstrated on the pizza line. In this case it involves processing of multiple instructions, each in a different pipeline stage. This is analogous to the pizza line where an instruction corresponds to a pizza and every processor stage corresponds to the pizza line stages. Similar to the pizza line, in this case the throughput is improved while the latency of an instruction execution may increase. Let's assume that latency of each processing phase is summarized by Table 7.

In this case, the total processing latency of the single-cycle processor is 4 ns, i.e., 250 MHz clock frequency. The throughput in this case is 1 instruction per 4 ns. If every processing phase is implemented as a pipeline stage, and instructions move from one stage to another ideally every clock cycle, the clock cycle time can be set to 1 ns (determined by the slowest stage). Assuming instruction can ideally move from one stage to another every clock cycle, the pipelined processor throughput becomes one instruction per cycle and thereby gaining 4× throughput improvement with respect to the single-cycle processor. It can be observed that the processing latency of the pipelined processor does not improve but the contrary. The pipelined processor latency is 5 ns, while the single-cycle processor latency is 4 ns. This is similar to the prior observation in the pizza line example, where pipelining improves throughput rather than latency (Fig. 13).

Given, that pipelining is a great idea to improve the processor throughput, the next step will be to redesign the single-cycle core such that it processes instruction in a pipelined manner. Luckily, the design modification needed to the single-cycle data path are relatively simple as depicted by Fig. 14. Four additional registers are added to sample the data being transferred between two adjacent pipeline stages. Another modification is required for the register identifier, dreg, in the write port of the register file. Unlike the single-cycle processor where the entire processor logic processes the same instruction, in the pipelined processor, different pipeline stages process different instruction phases concurrently. As a result, keeping the dreg signal as it was connected in the single-cycle processor design, the result from the write-back stage may be written to an incorrect register. This may happen because the destination register in the write-back stage can be different from the one in the decode stage. The solution to this problem is relatively simple. The destination register identifier dreg is sampled by every pipeline stage starting from the decode

Table 7 Latency of execution phases

Phase	Latency [ns]
Fetch	1
Decode	0.5
Execution	1
Memory	1
Write-back	0.5

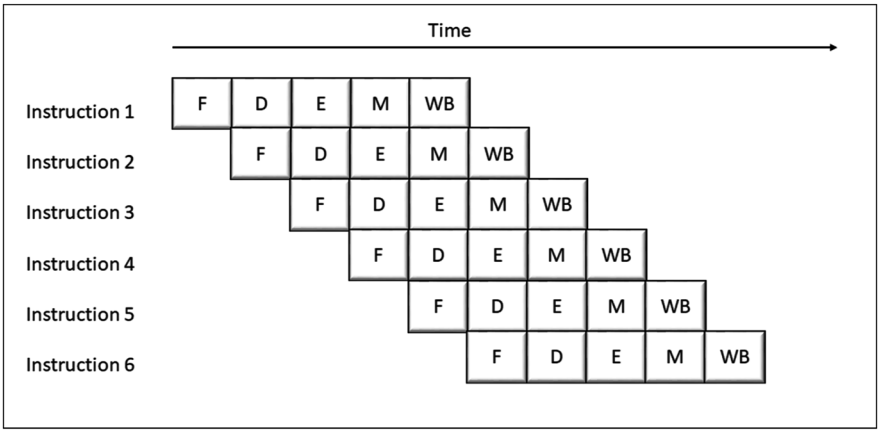


Fig. 13 Pipelined instruction processing

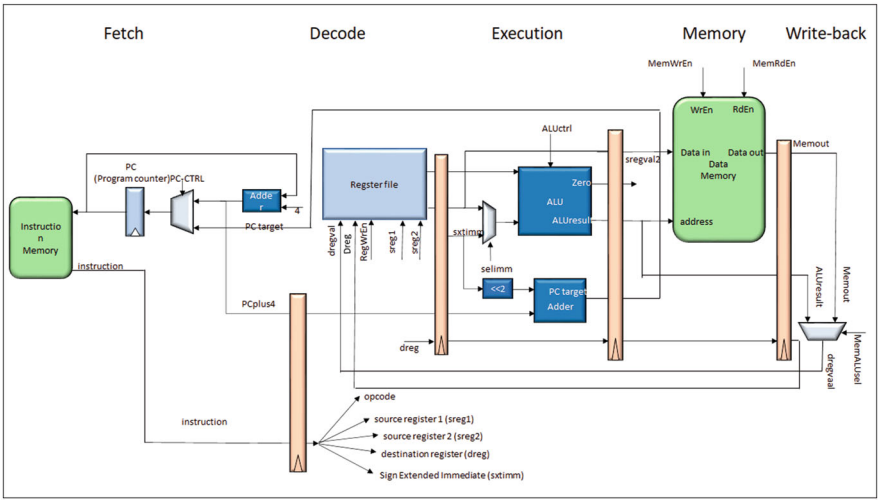


Fig. 14 Pipeline processor data path

stage and is used by the write-back stage as the identifier for the register file write port.

The pipeline processor design also requires some slight modifications in the control unit. Recall that the control unit is implemented as a combinatorial circuit using the opcode and zero signal inputs to generate the needed control for the processor data path. Since the opcode is available for the control unit at the decode stage, the output control signals of the control unit need to be retimed to their corresponding pipeline stages. The retiming of the controls signals is presented by Fig. 15: The ALUctrl is retimed with one sampling stage to the execution stage.

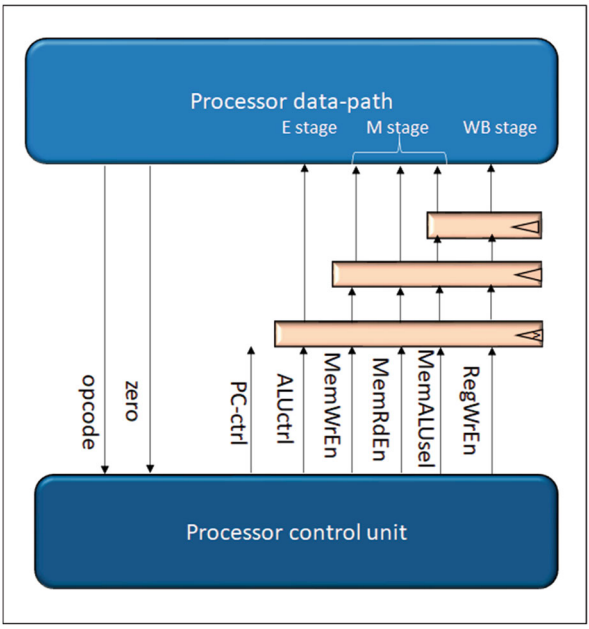


Fig. 15 Control unit in pipelined processor

The MemWrEn and MemRdEn are retimed to the memory stage with two sampling stages. Finally, the MemALUSel and RegWrEn are retimed to the write-back stage with three sampling stages.

Now that the five-stage pipeline core has been established, one can exercise the processing of the following code through pipeline stages depicted by Fig. 16. The pipelined processor achieves the maximum possible throughput of one instruction completion per clock cycle. Is this always the case? Can the processor achieve the maximum possible throughput for every program? This question will lead to the next discussion.

Pipeline Hazards

By taking a deeper look at the pipelined processor, it is possible to identify that its pipeline implementation may involve pipeline hazards. A pipeline hazard is defined as a situation in the pipeline that may lead to an incorrect execution of an instruction. Pipelined processors may have three types of pipeline hazards as the following:

- Data hazards
- Control hazards, and
- Structural hazards

Pipeline hazards are the next focus of the discussion: Data hazards may occur due to incorrect data transfer between instructions, control hazards are related to

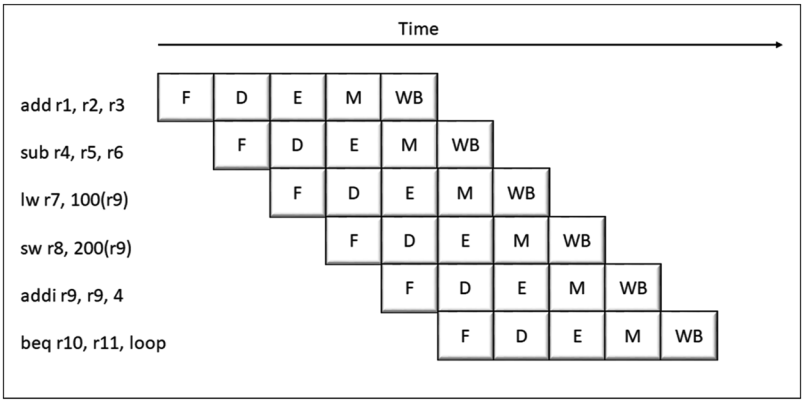


Fig. 16 Instruction execution in pipeline processor

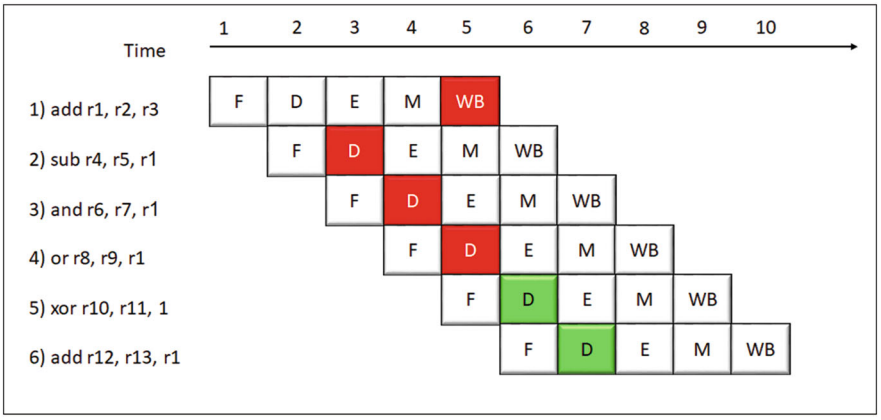


Fig. 17 Pipeline execution with data dependencies

any control flow operation such as branch or jump, while structural hazards are related to pipeline hardware design conflicts.

Data Hazards

In order to further examine the pipeline processor, one may consider the following code.

As it can be observed by Fig. 17, instruction 1 writes to r1 while all the successive instructions read r1. It can be observed that instruction 1 write r1 value in the register-file only at clock cycle 5 while instructions 2, 3 and 4 read r1 before it is updated in the register-file. Only instructions 5 and 6 in the example above read the correct data of r1. This type of hazard is termed as a read-after-write (RAW) hazard or true-data dependency. True-data dependency (or RAW hazard) occurs when an instruction reads an outdated source operand because it has not been calculated

or written yet. This may lead to incorrect program execution, and therefore the next discussion will delve into various microarchitectural solutions to this problem. RAW hazard is the only type of data hazard that may occur in a single pipeline processor. In the future discussions on multiple-issue processors, additional types of data hazards will be introduced. Typically, RAW hazards are associated with data that is transferred through register containers, though, theoretically, they can also occur when data is transferred through memory. In a pipelined processor, RAW hazards can only be related to registers rather than physical memory elements, since all read and write memory accesses take place at the same pipeline stage, the Memory stage. How can data hazards be solved then? The simple method is to have the compiler insert no-operation (nop) instructions between true-data-dependent instructions. This is illustrated by Fig. 18.

Nop instruction insertion method has a major impact on pipeline utilization. As it can be observed by Fig. 18, such an approach reduces pipeline utilization and increases the effective CPI of the processor. Instruction scheduling, illustrated by Fig. 19, is another approach to handle data hazards. In this case, the compiler (or the programmer) reschedules instruction order within the program (while preserving the program correctness) with independent instructions. Instruction scheduling can significantly help improving the pipeline utilization if independent instructions can be found for rescheduling. The limitation of such a technique is mainly due to the fact that it is performed at compile time where the pool of candidate-independent instructions for rescheduling may be limited due to the lack of run-time information (such as branch direction, etc.).

A similar approach to the nop insertion is a hardware-based interlock mechanism which dynamically detects the data hazards and generates pipeline stalls which performance-wise are equivalent to the nop insertion (Fig. 20).

Since this is implemented by the hardware, the compiler is dismissed from the process of nop insertion, and therefore the program footprint size is smaller. A

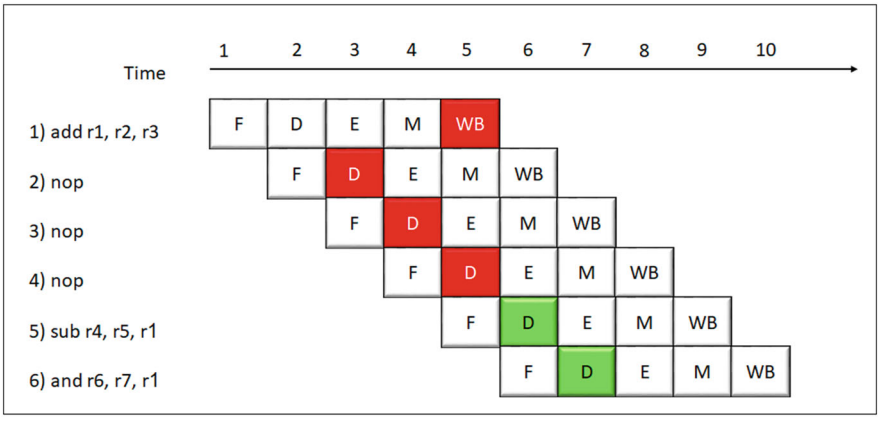


Fig. 18 Data hazards avoidance by nop insertion

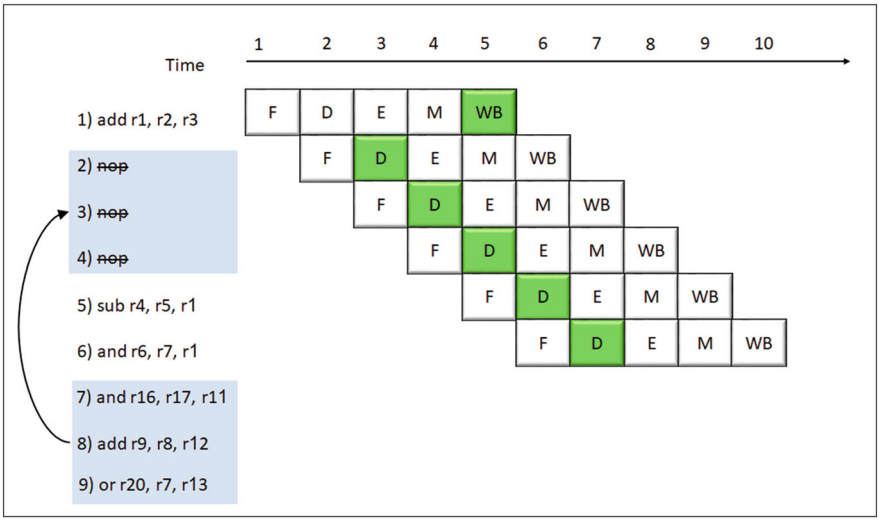


Fig. 19 Instruction rescheduling

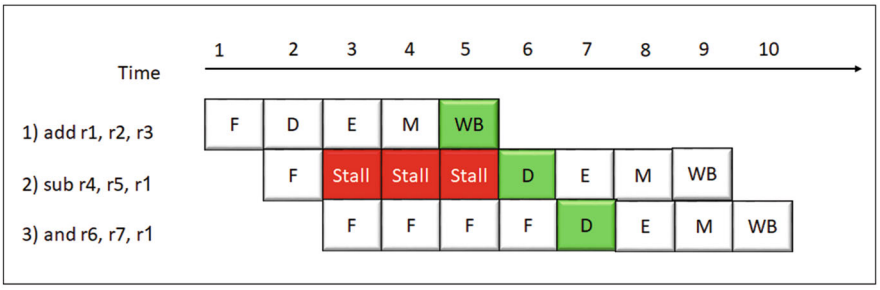


Fig. 20 Pipeline interlock for data-dependent instructions

pipeline interlock scheme typically consists of two logical functions: (1) RAW hazard detection and (2) Stall insertion, as illustrated by Fig. 21. The RAW hazard detection circuitry performs comparisons of the source registers (sreg1 and sreg2) of the instruction in the decode stage with the destination register (dreg) of the instructions at the execute, memory, and write-back stages. In case of match, a stall-generation signal is asserted. The stall-generation signal is connected to the processor control unit which overrides the control signals generated for the decoded instruction (presented by Tables 2, 3, and 4). In the case when the stall generation is true, the control signals are overridden with the values presented by Table 8. These overridden values are equivalent to nop-operation instruction since they keep the architectural state of the process unchanged. The overridden control signals will continue to be inserted into the pipeline till the stall generation signal becomes false.

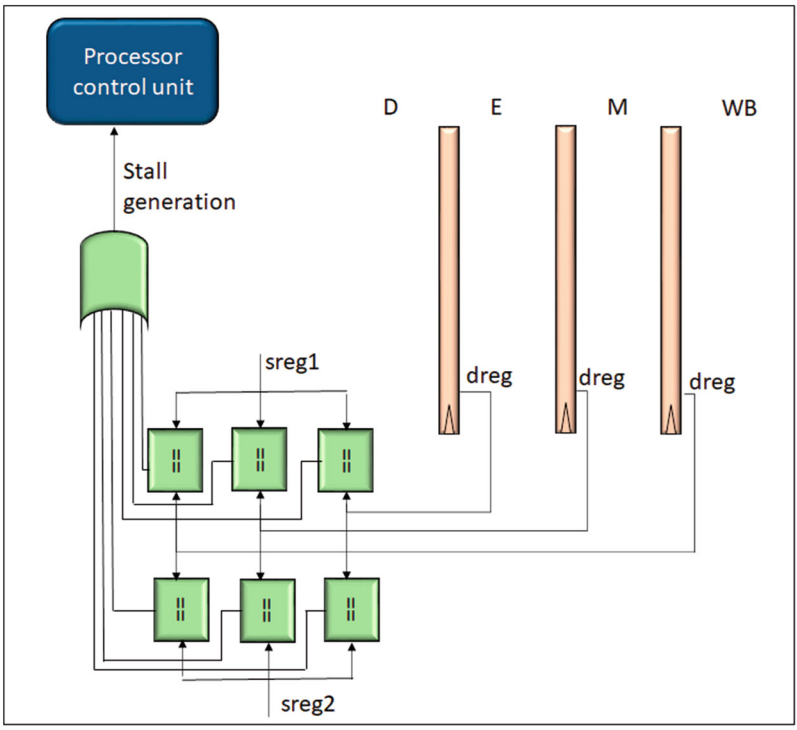


Fig. 21 Pipeline interlock circuit

Table 8 Modified control unit to support stall generation

Opcode	Stall generation	Zero	PC ctrl	ALUctrl	MemWrEn	MemRdEn	RegWrEn	MemALUsel
X	1	X	X	nop	0	0	0	0

Nop insertion or stall generation introduces high-performance overhead due to the impact on the pipeline utilization. In the next discussion, a hardware-based technique, called forwarding, is introduced, which minimizes the impact of true-data dependencies.

The concept of forwarding relays on the observation that the main cause for the performance overhead related to RAW hazards is due to the fact that data is transferred between instructions through registers or any other temporary storage elements. Since registers are read and written in different pipeline stages, a read operation of data-dependent instruction is forced to wait till the write with the most updated value is completed even if the value is already known earlier. For example, looking at the example presented by Fig. 16, the result of the first instruction is already known when it completes the execution stage, but the write to the destination register takes place only at the write-back stage. Another observation is that the

second instruction, which is data dependent on the first one, needs the value of r1 only when it starts execution. If one can bypass the register file-based data transfer between instructions, these data hazards and stalls may be eliminated. The concept of the forwarding idea relies on this principle by employing a bypass network (also termed forwarding network) which allows instruction to transfer date while bypassing the register file (this technique is also sometimes referred to as bypassing). In order to implement the forwarding mechanism, the first step is to modify the register file implementation (which is presented by Fig. 4). The new implementation, illustrated by Fig. 22, allows write and reads of the same register to take place at the same clock cycle. This is done by adding two bypass multiplexors and two comparators. Each of the comparators compares the destination register identifier with the source register identifiers. In case of match, the multiplexors replace the original value read from the register file with the value of the register being written. This scheme helps to eliminate one out of the three stall cycles in the pipelined processor, as illustrated by Fig. 23, since write and read operations to the same register are now allowed to take place at the same clock cycle. In order to further eliminate the additional pipeline stalls and improve pipeline utilization, the forwarding network illustrated by Fig. 24 is introduced.

The illustrated forwarding network monitors the destination register identifiers of instructions which completed their execution or memory stage and compares it with the sreg1 and sreg2 register identifiers of the instruction that entered the execute stage. In case of match, the most update value of the corresponding register is replaced by the 3-to-1 multiplexors illustrated by the figure above. It should be noted that the forwarding logic complicates the pipeline design. Additional hardware is added (comparators, multiplexors, and interconnect wires) which may increase

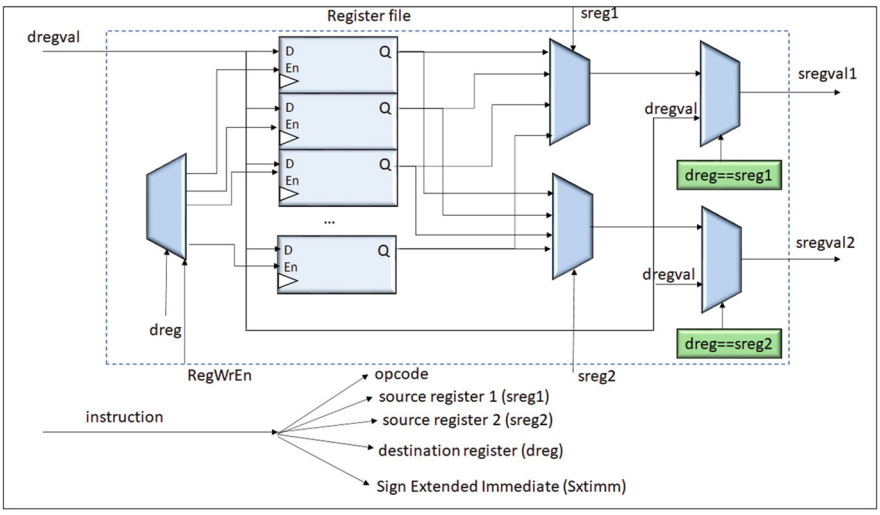


Fig. 22 Register file scheme with forwarding support

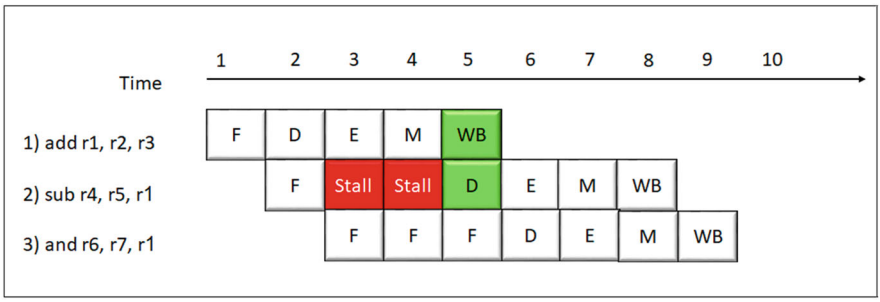


Fig. 23 RAW hazards with register-file forwarding

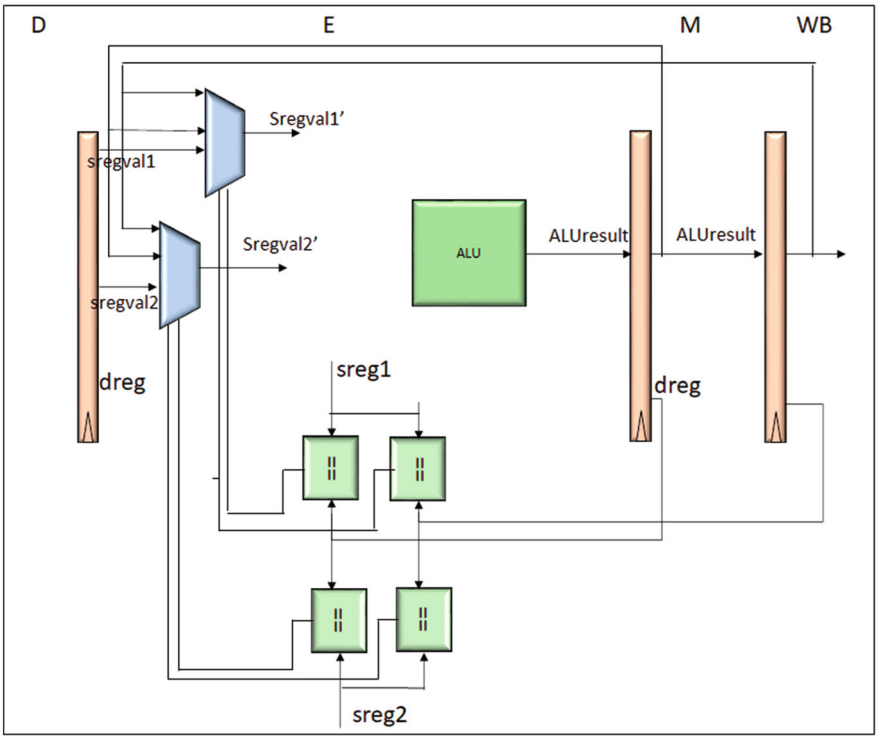


Fig. 24 Forwarding network

silicon area and power. In addition, it may affect the critical path of the logical circuit, resulting in reduced clock frequency. Another implementation option for the forwarding network as part of the microarchitectural considerations is to retime the comparators and move them from execution stage to the decode stage as illustrated by the following figure (Fig. 25).

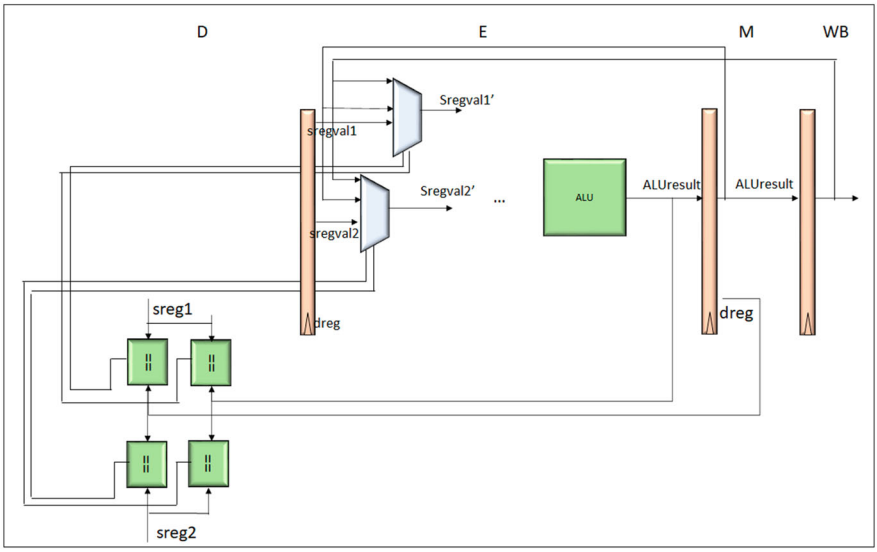


Fig. 25 Forwarding network with retimed comparators

An important factor which affects the forwarding network complexity is the pipeline depth. Deeper pipeline structures, known as super-pipelines, will incur much more complex forwarding network involving more comparators and wider multiplexors. This can, off course, affect silicon area, power, and timing. Now that the forwarding design is implemented, the example program from Fig. 17 can be rerun in order to demonstrate how the previous stall cycles can be avoided, thereby achieving CPI=1 (Fig. 26).

The next question is whether the proposed forwarding scheme has been able to avoid all possible RAW hazards. Have all scenarios been handled? The answer to these questions is that there is one more additional scenario to take into account in the pipelined processor with respect to forwarding. This scenario is related to load instructions. Unlike ALU instructions which complete the computation of their result at the execution stage, load instruction will require an additional clock cycle for the memory access stage. As a result, load instructions cannot forward their outcome value from the execution stage but only from the memory stage. Therefore, a dependent instruction on a load may incur one clock cycle of pipeline stall as illustrated by Fig. 27. Such a stall may be eliminated of course if the instruction rescheduling technique, which was previously presented, is employed and find an independent instruction to be scheduled after the load to hide the pipeline stall.

Control Hazards

Control hazards, also known as control dependencies, occur in pipelined processors whenever the control flow of the processor is changed. Ideally, if computer programs could be written without jumps or conditional branches, control hazards would be

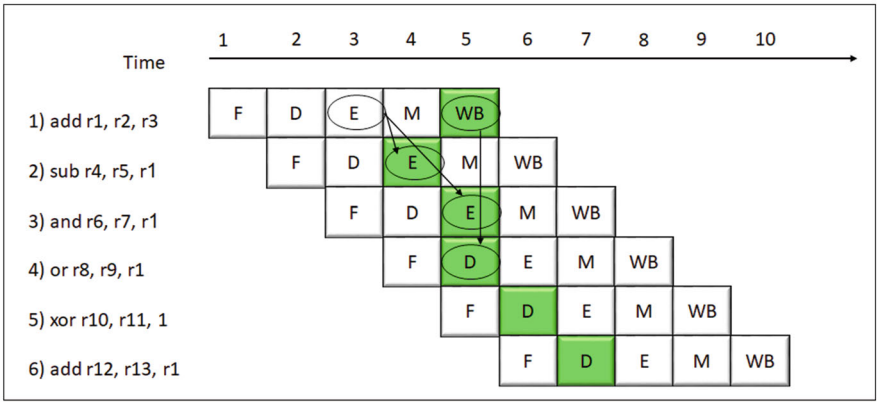


Fig. 26 Fully utilized pipeline execution with forwarding

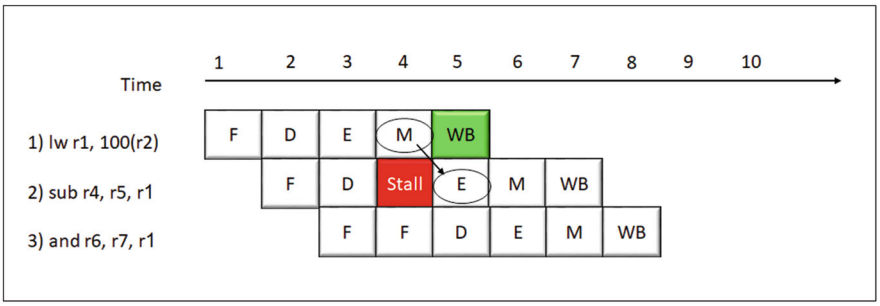


Fig. 27 Load instruction RAW hazard and forwarding

avoided. For the sake of simplicity, three types of control flow instructions are considered:

- Direct conditional branches, e.g., beq r1, r2, loop
- Direct unconditional jump, e.g., j label
- Indirect unconditional jump, e.g., jr r1

All three types of control flow instructions can change the program counter and disrupt the sequential instruction fetch process. Unfortunately, the processing of such instructions cannot take place immediately as soon as they enter the pipeline since they require processing which takes several clock cycles until their resolution. The resolution process of control flow instruction involves identifying the instruction type, calculating the target address, and evaluating the branch condition for conditional branches. As a result, as long as the outcome of such control flow instructions is not resolved, the next instruction to be fetched into the pipeline cannot be determined. In the pipelined processor design, illustrated by Fig. 14, the resolution of control flow instructions occurs at the memory stage where the PC control is generated, and the target address updates the program counter.

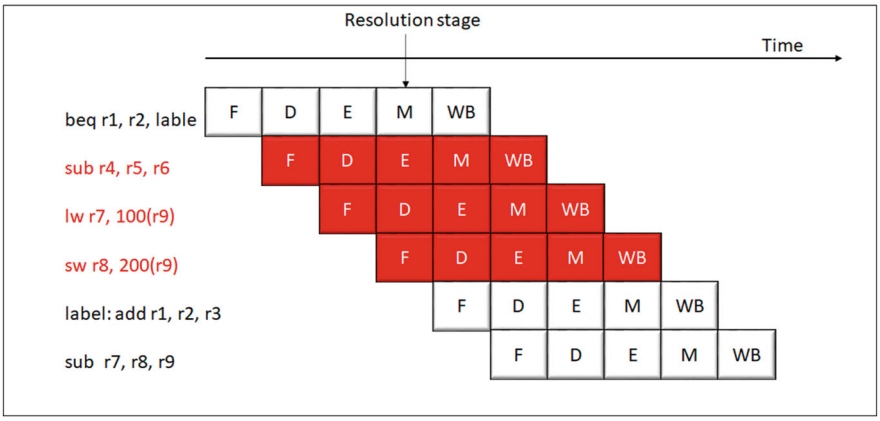


Fig. 28 Control hazards in pipelined processor

This phenomenon, known as control hazard, may result in incorrect execution due to consecutive instruction fetch until the resolution. This situation is illustrated by Fig. 28 where it can be observed that in the case that the conditional branch is taken, three consecutive instructions are fetched and executed till the branch processing is resolved (memory stage).

One of the possible solutions for control hazards is to stall the pipeline in a similar fashion to data hazards by employing the pipeline interlock mechanism. This may guarantee the correct execution of a program, but it will lead to a major performance degradation since control flow instructions are quite frequent. For example, assume a program with an ideal CPI=1 and branch frequency of 20%. Every branch instruction in the presented processor will involve three stall cycles until resolution. Therefore, the effective CPI becomes 1+0.2*3=1.6. This is a major performance degradation of 37.5% in respect to the ideal CPI. It can also be noted that the deeper the pipeline is, the greater is the branch penalty. For example, by breaking every pipeline stage of the processor into two stages, the branch resolution stage is now at the seventh or eighth stage. Such a pipelined processor will incur 6–7 clock cycle penalty for every control flow instruction.

Moving forward, while keeping Amdahl's law in mind ("make the common case run faster"), there is an essential need for effective solution to guarantee correct program execution while improving utilization of the pipelined processor. Toward that direction, the first step to handle control hazards is to reduce the branch penalty. This can be done by moving the branch resolution stage to an earlier pipeline stage. In the current processor design, the branch resolution takes place at the memory stage. Assuming that the branch resolution can move to the decode stage, the branch penalty can be reduced from three clock cycles to one clock cycle. The required microarchitectural changes in the pipeline data path are depicted by Fig. 29 in a light gray color. First, the adder that calculates the target address is moved from the execution stage to the decode stage. Next, a new comparator is added to compare to

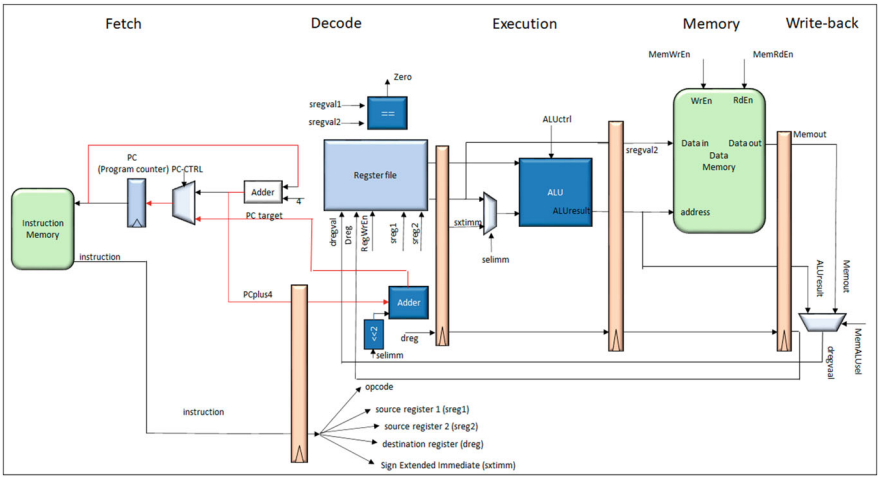


Fig. 29 Pipeline data path with branch resolution at the decode stage

source register values. This was previously done by the ALU at the execution stage, and since the ALU can be busy processing the instruction in the execution stage, it can no longer be considered to do this task. There are several important implications related these changes: (1) The added logic increases die area and power. (2) The logical path calculating the PC target becomes much more stressed for timing point of view. This is because at the same clock cycle, the PC is incremented by 4, the sign extended immediate is added, and the target address is sent through a multiplexor to the PC register. This timing path is shown in red color in Fig. 29. From the design point of view, assuming that this path is not the critical path of the processor (ALU and memory-related logical path typically take longer processing time), this will still require using faster logical elements and as a result may affect the processor power.

Figure 30 illustrates the control hazard when branch resolution is moved to the decode stage. It can be observed that it is possible to reduce the control hazard penalty in the processor from three cycles to one cycle.

The proposed design changes in the processor to perform the branch resolution at the decode stage affect the forwarding mechanism. So far, the assumption is that the forwarded values are needed at the execution stage; now, de facto, the processing of control flow instructions is performed at the decode stage. This implies that when the branch instruction is true-data dependent on a predecessor instruction, the branch will be stalled at the decode stage for one clock cycle till the dependent source value can be forwarded. This situation, illustrated by Fig. 31, requires two design modifications in the pipelined processor:

- The forwarding network will have to be changed such that it will be able to perform forwarding to control flow instructions at the decode stage. This will

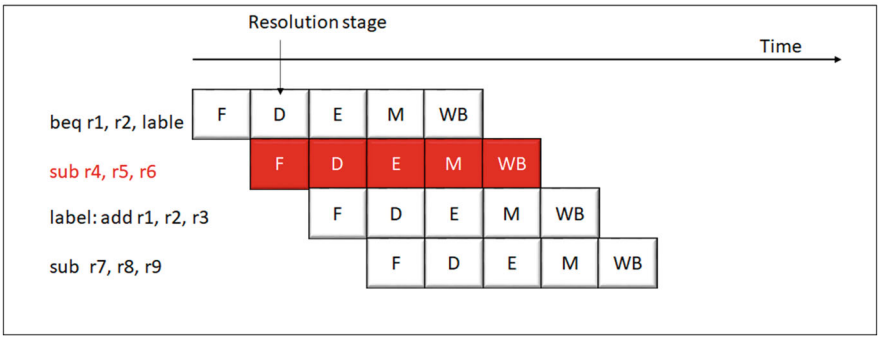


Fig. 30 Control hazards when branch resolution moves to decode stage

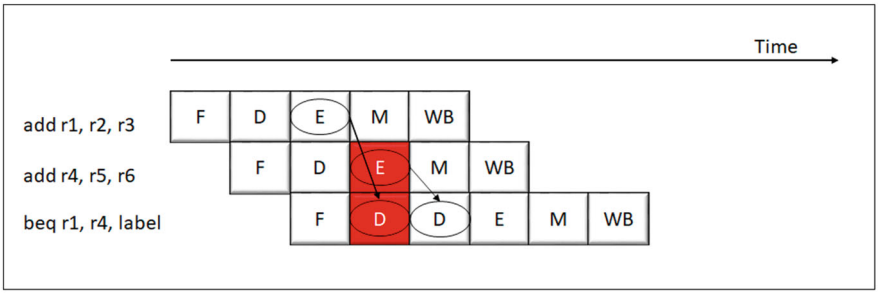


Fig. 31 RAW hazards associated with branch instruction processed at the decode stage

add additional comparators and multiplexors which may stress timing paths and can increase cycle time and power.

- The pipeline control unit will have to detect this situation and stall the branch for additional clock cycle at the decode stage.

The next improvements for handling control hazards are based speculation and also known as branch prediction. The principle of branch speculation relies on two fundamental mechanisms: (1) A branch predictor and (2) A pipeline flush mechanism which can flush all mis-speculated (mis-predicted) instructions which follow the branch. As long as the prediction is correct, the branch penalty is saved; however, in case of mis-speculation, the flush mechanism is activated to invalidate all the instructions from the mis-predicted path. The flush mechanism is usually implemented in the microprocessor control unit. Upon detection of branch mis-prediction, the control performs the flush by invalidating all the control signals of the instructions in a similar fashion to the stall generation described earlier.

Before moving forward with the branch predictor discussion, the performance metrics to be used for quantitative performance evaluation of branch prediction schemes are defined. The first metric, branch mis-prediction rate (MR), is defined as

$$MR = \frac{Mis - predicted\ branch\ instructions}{Total\ number\ of\ branch\ instructions\ executed} \tag{2}$$

The second metric, mis-predictions per instruction (MPI), is given by the following formula, which takes into account the branch occurrence rate:

$$MPI = \frac{Mis - predicted\ branch:\ instructions\ instructions}{Total\ number\ of\ instructions\ executed} \tag{3}$$

There are two types of branch prediction mechanisms: static and dynamic. Static branch prediction usually provides a constant prediction of not taken and continues fetching instructions in a sequential manner. Once the branch is resolved if the prediction was found to be correct, then the branch penalty is avoided; however, if the branch is mis-speculated (or mis-predicted), all the instructions for the mis-speculated path are flushed. Static prediction of taken is more complicated to implement since the branch target is unknown at the fetch stage. Dynamic branch predictors attempt to predict both branch direction and target address dynamically by learning branch behavior based on their history. The most common implementation of dynamic branch predictor is termed Branch Target Buffer (or BTB), which is illustrated by Fig. 32.

The BTB is organized in a cache structure. Each BTB entry consists of Tag, valid bit, target address, and a history bit. The look-up process in the BTB is performed usually at the fetch stage and is similar to the cache memory look-up process. The index field from the PC selects a set, while the tag field is compared against the tag in the entries of all the ways within the set. In case one of the tag matches and valid=1, the look-up process achieves (BTB hit), when the target field and the history bits

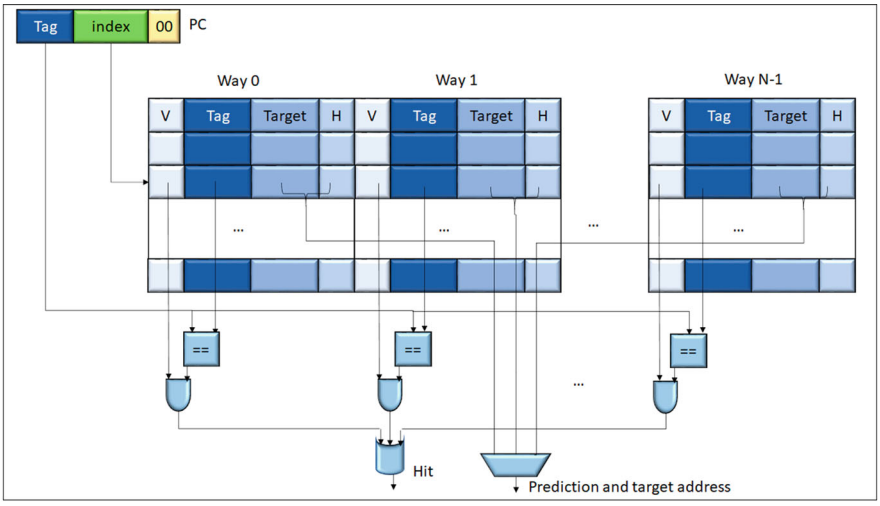


Fig. 32 Branch Target Buffer

are read from the corresponding matched entry. The history bit is used to determine the branch direction. If the history bit is 1, the branch will be predicted as taken; otherwise, the branch will be assumed not taken. When the branch is predicted as taken, the target field will be loaded to the PC in the next clock cycle. The fetch stage will continue fetching instructions from the target address in a consecutive manner. In case of BTB miss (PC is not found in the BTB), the fetch stage continues the instruction fetch in a sequential manner. Once the branch is resolved, the following action takes place:

- The BTB is updated based on the branch resolution: both history and target fields.
- In case of branch mis-prediction, the instructions from the wrong branch path are flushed from the pipeline and the PC is loaded with the correct address.

Due to the major impact of control dependencies on processor performance, BTB microarchitecture has been in the focus of both industry and research community in the past decades. Various BTB schemes were suggested, and the main ideas are presented in the following discussion. Lee and Smith (1984) suggested two-bit saturated counter to replace the single bit of history in the BTB. The two-bit saturated counter is illustrated by Fig. 33 state diagram, which employs four states. When the counter is in states 00 (strongly not taken – SNT) or 01 (weakly not taken, WNT), the prediction is not taken, while for states 10 (weakly taken, WT) and 11 (strongly taken, SNT), the prediction is taken. The prediction is determined based on the most significant bit of the state. The state transitions, illustrated by Fig. 33, suggest that upon every taken branch, the state is incremented (saturated in 11), while for every not taken branch, the state is decremented (saturated in 00).

For example, let us consider the following code, presented by Fig. 34, to evaluate the performance of the two-bit saturated counter predictor. Also assume that the initial state of the counter is WT. As it can be observed by Fig. 34, the first branch

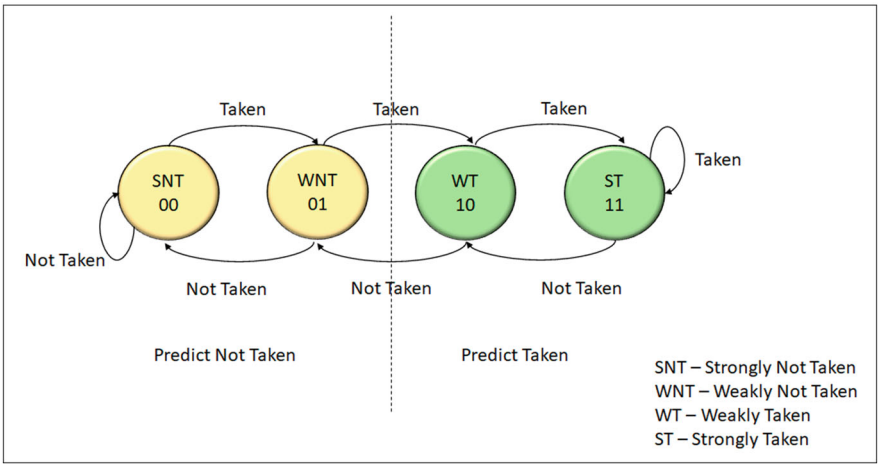
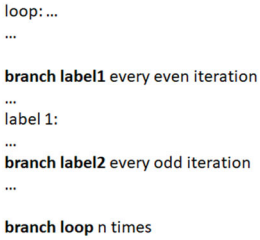


Fig. 33 Two-bit saturated counter BTB



Branch label 1							
Actual	T	NT	T	NT	T	NT	...
Predictor	NT	T (WT)	NT (WNT)	T (WT)	NT (WNT)	T (WT)	NT (WNT)
BTB hit/miss	Miss	Hit	Hit	Hit	Hit	Hit	Hit

Branch label 2							
Actual	NT	T	NT	T	NT	T	...
Predictor	NT	T (WNT)	T (ST)	T (WNT)	T (ST)	T (WNT)	NT (ST)
BTB hit/miss	Miss	Hit	Hit	Hit	Hit	Hit	Hit

Branch loop							
Actual	T	T	T	T	T	...	NT
Predictor	NT	T (WNT)	T (ST)	T (ST)	T (ST)	T (ST)	NT (ST)
BTB hit/miss	Miss	Hit	Hit	Hit	Hit	Hit	Hit

Fig. 34 Two-bit saturated counter branch predictor example

instruction is always mis-predicted, i.e., the MR=100%, while in the second the predictor gains MR of nearly 50%. The last branch instruction is predicted correctly in all iterations except the first and the last iterations. More advanced branch predictors, termed two-level branch predictors (Yeh and Patt 1991), perform their prediction in two stages. The types of predictors can be classified into two groups: local predictors and global predictors. Local two-level predictors perform their prediction based on local (private) past history of every branch instruction, while global two-level predictors use the global history of recent branches executed for the prediction process.

The baseline scheme of the two-level local predictor, depicted by Fig. 35, expands the one-bit history field shown in Fig. 32 to multiple history bits. An n-bit history field, termed BHR (Branch History Register), represents the local history of the corresponding branch. For example, for n=3 a history of 101 represent a branch that was taken, not taken, and then taken. The n-bit BHR field is used as an index to address the second level of the BTB which consists of sets of saturated counters. The prediction is determined based on the state of the counter which corresponds the BHR value. In the baseline scheme, every branch history is associated with a private set of saturated counters, but since this may introduce expansive die area cost, various compromises were suggested. For example, in the Pentium III two-level BTB, all branches within the same BTB set share the same set of counters. The Alpha 21264 (1999) shares one set of counters with all BTB entries. Other schemes attempt to minimize the interreferences between conflicting branches that share the same set of counters. For example, the low-significant bits taken from the program counter can be concatenated with the BHR as index to the shared counters. Other approach suggests performing a bitwise xor between the BHR and n bits taken

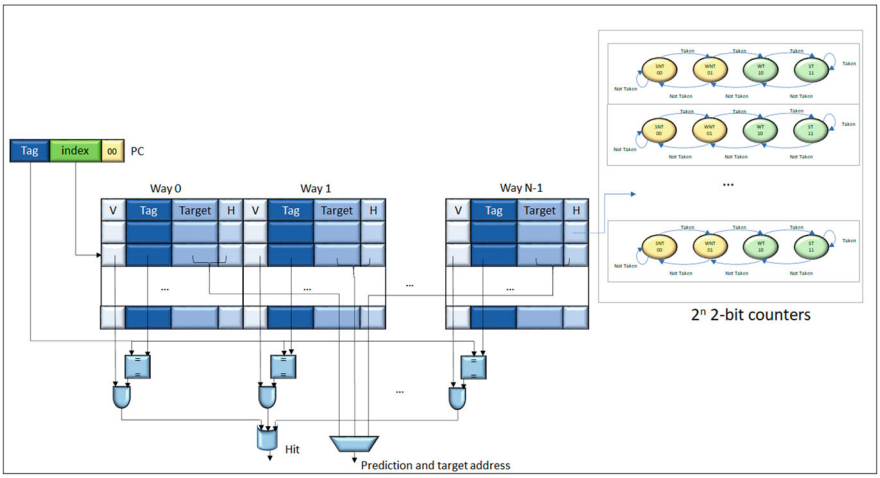


Fig. 35 Two-level local BTB

from the PC. Both schemes reduce the likelihood for collisions by using some bits from the PC to map with the same history to different counters.

Global two-level branch predictor replaces the local BHR fields with one global history register (GHR) (Mittal 2019; McFarling 1993). An n-bit GHR represents the history of the n recently executed branches in the program. In the baseline scheme of the two-level global BTB, presented by Fig. 36, the n-bit GHR is used as index to a set of 2n saturated counters which determine the prediction per every different combination of global history. One of the potential problems with the global predictor is the case of uncorrelated branches which exhibit different behavior for the same history. This may prevent the two-bit counters to correctly predict the branch outcome. One of the solutions to this problem, termed g-share (Skadron et al. 1998), suggests performing a bitwise xor on the GHR with bit from the PC. This may help to spread different branch instructions with the same history to different saturated counters.

Various processors, such as Alpha 21264 (1999), employ hybrid predictors (also termed tournament predictor) which combine both local and global two-level BTBs. A general structure of a hybrid predictor is illustrated by Fig. 37. The hybrid predictor is governed by a chooser mechanism. The chooser learns per every prediction which BTB is the preferred for the prediction. For example, in the Alpha 21264, the chooser is implemented as an array of two-bit elements indexed by the GHR. Every two-bit entry represents a different predictor local or global. A value of 1 indicate that in the last prediction, the corresponding predictor was correct; otherwise, the predictor was wrong. The chooser in this case uses the local predictor only if in the last prediction which corresponds to the GHR value, the local was correct and the global was wrong. In all other cases, the global predictor is preferred. There may be different implementations for the chooser. For example, the chooser

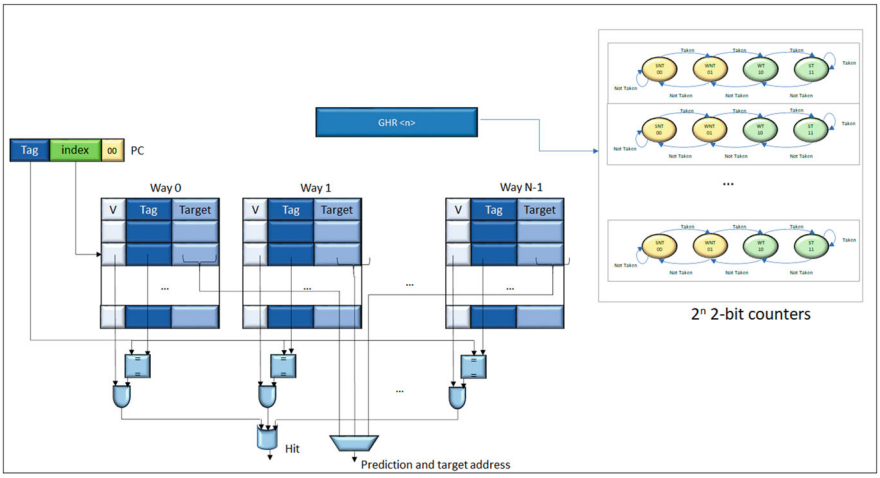
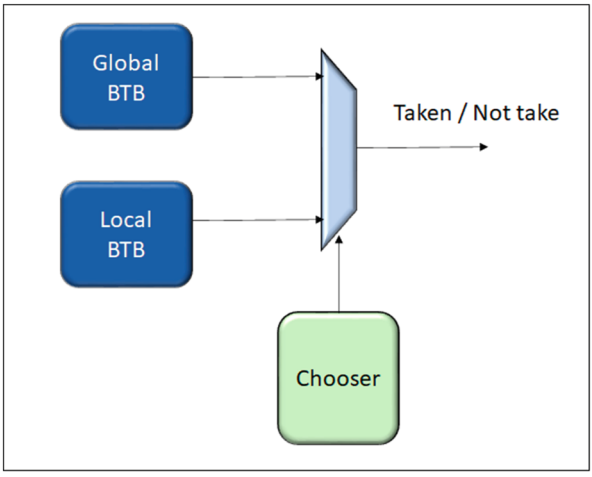


Fig. 36 Two-level global BTB

Fig. 37 A hybrid BTB



array can be indexed by the PC instead of the GHR; in addition, the chooser array may also implement various FSMs to learn the preferred predictor per every branch.

In deep processor pipeline design, updating the BTB can be become a highly complicated process due to the long latency between the fetch stage, where the BTB lookup is performed, and the branch resolution stage where the BTB update occurs. As a result, new branch instructions may enter the processor pipeline while the prior branches are not resolved. If the history and the counters are not updated till the branch is resolved, the new branches may see outdated history and counter state resulting in high mis-prediction rate. Therefore, the BTB is speculatively updated with the prediction; however, in case of mis-prediction it is needed to roll back to

the BTB state prior to the speculative updates. A special hardware mechanism is needed in the BTB both for maintaining the speculative updates and for recovering the BTB state in case of mis-prediction. Additional advanced branch prediction mechanisms can be found in modern processors, such as the return stack buffer (RSB) (Skadron et al. 1998) which is used to predict the target address of a return from subroutine, the loop predictor which is used for the branch loop prediction, and the iBTB (Simcha Gochman et al. 2003) used for indirect branch prediction.

Structural Hazards

Structural pipeline hazards occur due to either lack of hardware resources or in the case of a collision on resources needed by multiple pipeline stages. There can be several scenarios of structural conflicts. For example, if assume a unified cache memory for both instruction and data, then both fetch stage and memory stage may access the cache at the same clock cycle. If the cache has a single access port, then this is a structural hazard that can be resolved either by using an arbiter for the cache access or by duplicating the cache port to solve the collision. Another solution which was adopted by the processor was a split instruction and data caches, also known as the Harvard architecture. An additional example of structural hazard happens if it is decided to retime the write-back to the register file and perform writes of non-load instructions at the memory stage instead of the WB stage. Loads write-back are kept to occur at the WB stage. The structural hazard in this example happens when an ALU instruction at the memory stage attempts writing to the register file while simultaneously a load instruction tries to write at the WB stage. Since the register file has only one write port, this structural hazard will have to be resolved by duplicating the write port of the register file or by adding an arbiter to arbitrate between writes from these two stages.

Multiple-Issue Processor

So far, the discussion was focused on a single pipeline processor where instructions were executed in order. Various techniques have been presented to improve pipeline utilization and overcome pipeline data hazards and control dependencies. What is the next evolution step for processors? Can performance be further improved? Recall that the execution time, provided by Equation 3, is a multiplication of the instruction count (IC), the average clocks per instruction (CPI), and the clock cycle time (T=1/f):

$$Execution\ Time = IC \times CPI \times T \tag{4}$$

Even if the presented pipeline is broken into smaller stages (super-pipeline), given the limitations of increasing clock frequency by the most of the advanced VLSI process nodes, this might be a limited option. Super-pipelining may also have several other limitations. Since they increase the latency between the fetch stage and the branch resolution stage, they incur greater branch mis-prediction penalty. In

addition, there is also a practical limit to the granularity of breaking the processor data-path and control unit into many pipeline stages. This will not only complicate the pipeline control unit, increase the number of sampling stages, and challenge the complexity of the forwarding network but is also limited by physical design consideration.

Another option to improve execution time is to reduce the IC. This approach, which was adopted by the CISC (Complex Instruction Set Computer) processors, suggests using complex instructions which can process multiple complex operations. This approach may eventually increase the clock cycle time and increase the CPI due to the complexity of the new instructions and the required logical circuits.

Decreasing the (CPI) or increasing the (IPC (instructions per clock cycle)) is also a valid approach to improve processor performance. In fact, all previous techniques presented in the previous section help to decrease processor CPI and improve pipeline utilization by minimizing pipeline stall cycles. CPI does not only depend on the processor microarchitecture but also on the program being run. For example, a program with high number of data dependencies may have low pipeline utilization and higher CPI relative to program with low data dependency rate. Can the CPI be further improved? Can one achieve CPI< 1 (or IPC>1)? In order to do so, the processor will need to handle multiple instructions being processed in parallel by multiple pipelines – such a type of processors is termed multiple-issue processors.

A (superscalar) processor, illustrated by Fig. 38, is a typical multiple-issue processor which employs multiple pipelines running in parallel. In this example, multiple instructions are fetched in parallel, decoded, executed, access the memory (if needed), and retire – all being processed in parallel manner.

The key for superscalar processor efficiency relies on the amount of parallelism exhibited by the program being run. The measure for such a parallelism, termed ILP (instruction-level parallelism), is defined as the average number of instructions which can be executed in parallel while preserving the program correctness. ILP is often illustrated using a dataflow graph, where every node represents an instruction

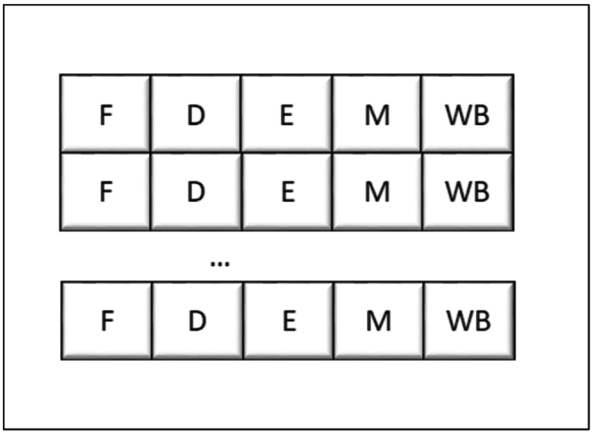


Fig. 38 A superscalar processor pipeline

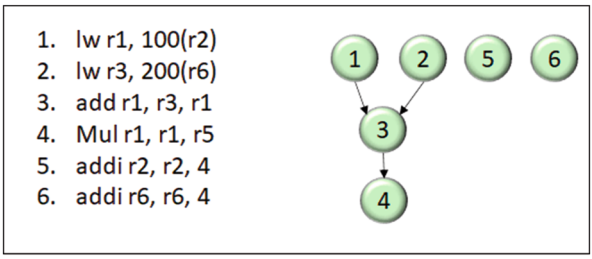


Fig. 39 A dataflow graph

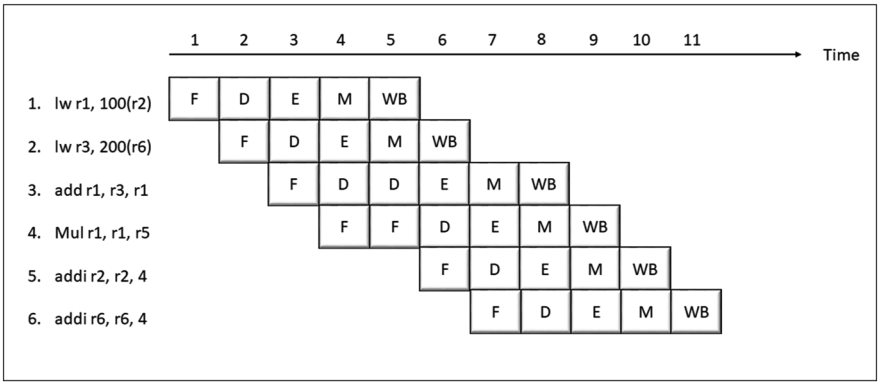


Fig. 40 Single pipeline execution

in the program and every vertex represents a true-data dependency as depicted by Fig. 39.

When running this code on a single pipeline processor (using the forwarding techniques previously presented), the total execution time is 11 clock cycles as illustrated by Fig. 40.

As illustrated by Fig. 41, when running the same code on a two-way superscalar processor, one may observe only a little improvement in the execution time which now becomes nine clock cycles. This is due to two main reasons: First the pipeline fill time is also counted as part of the execution time and in steady state this time will be become negligible. Second, it can be observed that due to data dependencies, the superscalar pipeline is not utilized efficiently. In clock cycle 5, two instructions complete, while in clock cycle 6 no instruction is completed and in clock cycles 7 and 8, only one instruction completes every clock cycle. This is concerning observation since the return-on-investment by duplicating the pipeline in the superscalar processors is not even close to x2 throughput improvement. The may reason for this limitation is because superscalar processor executes instructions in the program order, i.e., in-order execution, and as a result they have limited ability in exploiting the program ILP presented by the dataflow graph to improve pipeline efficiency. If the superscalar processor could be enhanced to run the program in the order of the dataflow graph (while preserving the program correctness), then the

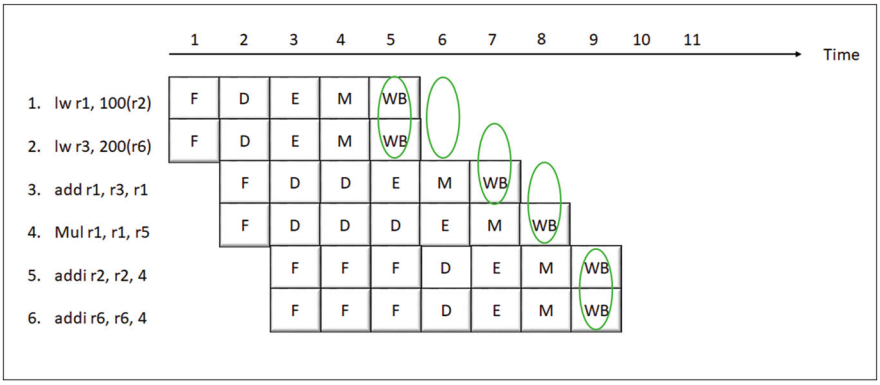


Fig. 41 Superscalar pipeline execution

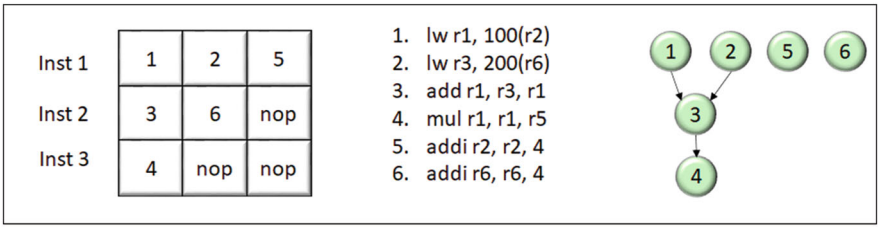


Fig. 42 Instruction scheduling in VLIW processor

pipeline efficiency will be significantly improved and would be limited by the ILP of the program.

Out-of-order execution is the process of executing instructions based on the dataflow graph while preserving the program correctness. There are two main approaches for out-of-order execution: static and dynamic. Static out-of-order execution relies on the compiler to reorder instruction in accordance with the dataflow graph. Very long instruction word (VLIW) processors rely on such an approach as illustrated by Fig. 42.

In VLIW processors, a very long word instruction container is used to encapsulate multiple instructions which can be executed in parallel. Since the encapsulation process is performed by the compiler, it may yield limited performance gain due to lack of information at compile time related to dynamic events which may cause pipeline stalls such as cache misses, branch mis-prediction, etc. Another limitation is that compilers cannot pack control-dependent instructions since the resolution of the branch is not known at compile time. VLIW processors, though, introduce several advantages. They simplify their hardware implementation, and as a result they can run in higher frequency or alternatively save power. For example, in such a processor, the need of forwarding network can be eliminated.

Dynamic out-of-order execution superscalar processors (Lipasti and Shen 1996) are commonly used by the industry. Such processors attempt to reschedule instruction execution at run-time in the order presented by the dataflow graph. Since instructions are executed in different order to the original program order, these processors are termed out-of-order processors. The principle of out-of-order superscalar processor relies on evaluating the program's instruction in a sliding window, known as the instruction window. Ready-to-execute instructions are fired to execution (assuming there are enough execution units) as soon as their source operands are ready (have been computed). Instruction computations will be committed to the architectural machine state (registers, memory, flags, etc.) in the original program order. This is essential due to several reasons. First, there is a need to preserve the precise order of interrupts or exceptions which are associated with instructions through their execution (e.g., page fault, divide by zero, etc.). Second, the sequence of architectural state changes should be visible to the user in an equivalent manner to an in-order processor. Last it is essential for maintaining pipeline flushes in case of branch mis-speculation. In the general scheme for out-of-order superscalar process, illustrated by Fig. 43, the processor pipeline consists of the following pipeline stages:

- Fetch (F) – Multiple consecutive instructions are fetched in parallel. BTB lookup is performed for control flow speculation.

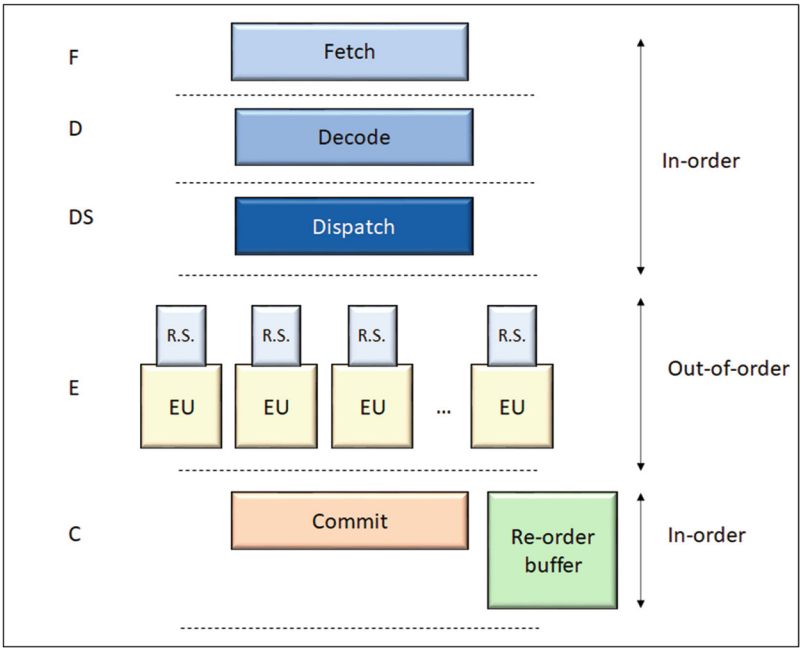


Fig. 43 A general scheme for out-of-order processor

- Decode (D) – Multiple instruction are decoded in parallel.
- Dispatch (DS) – Source operands are read from register file to all the instructions being processed.
- Execution (E) – Multiple instructions can be executed in parallel. Load and store instruction access the memory. Condition branches are resolved.
- Commit (C) – Instruction outcome results is written to processor's architectural state (memory or registers).

The fetch, decode, and dispatch stages process instructions in order, the execution stage executes instructions out-of-order, while the commit stage performs instruction retirement in an in-order manner. At dispatch stage instruction dependencies are typically evaluated and in case they are ready to execute, they are sent to the corresponding execution unit. True-data-dependent instructions cannot be executed and are pushed into the reservation stations. The reservation stations act as a buffer for instructions to wait till they become ready to execute. Every entry in the reservation station is connected to the forwarding network and snoops the results being forwarded. When all the source operands are ready, the instruction can be fired for execution. In the general scheme presented by Fig. 43, the reservation stations (R.S.) are implemented as distributed buffers attached to every execution unit. Various process implementations may use a unified reservation station which serves all execution units. At the execution stage, the calculated results are broadcasted to the forwarding network as soon as the result is ready and as a result may fire pending instructions for execution in the next clock cycle.

Out-of-order superscalar processors introduce major microarchitectural complexity and design challenges. The duplicated pipelines, complex pipeline control unit, and the dense forwarding network significantly complicate the processor design, verification, and physical design implementation process. In addition, they introduce additional types of pipeline data hazards known as write-after-write (WAW) hazards and write-after-read (WAR) hazards. WAW and WAR hazards, illustrated by Fig. 44, may lead to incorrect program execution. WAW hazard, also termed false dependency, occurs when two instructions that have the same destination register are reordered by the out-of-order processor. WAR hazard, also referred to as anti-dependency, happens when two instructions are reordered by the processor, and the later instruction in the program order writes to a register that is used by an earlier instruction. As a result, the earlier instruction may read the wrong value.

Can these anti-dependencies and false dependencies be solved? For example, renaming the detitanation registers in the examples above can eliminate these dependencies as illustrated by Fig. 45 while still preserving the correctness of the program.

Out-of-order superscalar processor employs register-renaming mechanism to eliminate both WAR and WAR hazards. The principle of register renaming is based on decoupling the architectural registers (which are part of the Instruction Set Architecture) used by the program code from the physical registers which can host them. As part of the register renaming scheme, the processor maintains a

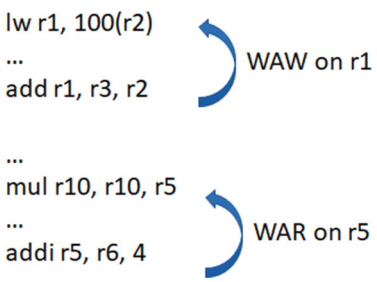


Fig. 44 WAR and WAW data hazards

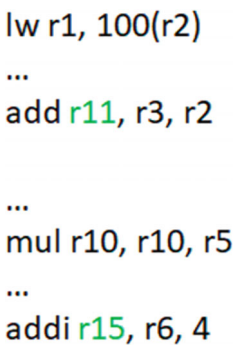


Fig. 45 Elimination of WAW and WAR hazards with renaming

bank of physical registers which can be larger than the number of architectural registers. In addition, a mapping table, termed register alias table (RAT), is used to map the architectural register identifiers to the physical register locations. The register renaming process, performed at the decode stage, involves replacing the architectural source registers with the physical register based on the mapping specified by the RAT. In addition, every architectural destination register is mapped to a new physical register and the RAT is updated. This mechanism assures that the processor core can eliminate both false and anti-dependencies as long as there are available physical registers. An example of register renaming process is illustrated by Fig. 46.

While register renaming can solve false and anti-dependencies, they cannot eliminate true-data dependencies (RAW hazards). True-data dependencies reflect the serial parts of the program code, and therefore the dataflow graph is considered as the upper bound on ILP. Various past studies suggested predicting the values being calculated by instructions and speculatively forwarding them to true-data-dependent instructions (Lipasti and Shen 1996; Gabbay and Mendelson 1997, 1998). This technique, known as value prediction, attempts to collapse true-data dependencies and exceed the dataflow limitations on ILP. If the prediction is found to be correct, instruction execution will continue with no disruption. In case of value mis-prediction, all the dependency chain which was fed by the incorrect prediction will be flashed out of the pipeline and re-executed using the correct value. Various value predictors were introduced such as the last-value predictor (Lipasti and Shen

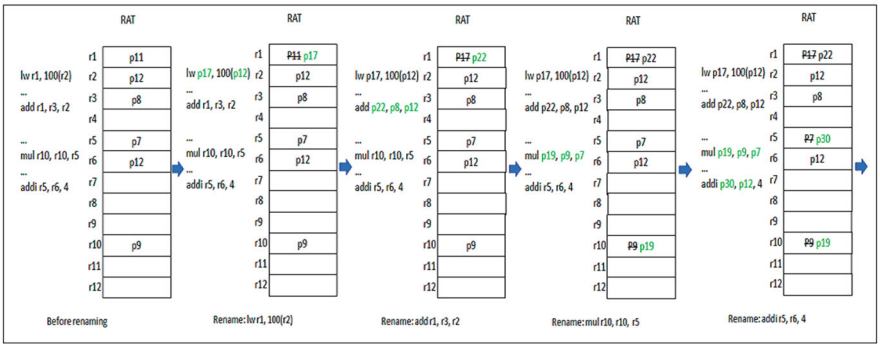


Fig. 46 Register renaming example

1996) and the stride value predictor (Gabbay and Mendelson 1998) and several others (Wang and Franklin 1999). Last-value predictor predicts the outcome value of an instruction based on the recently computed value by the instruction. Stride value predictor generalizes the last-value predictors and attempts to predict the destination value based on the last seen value plus a stride which is learned by the predictor. The stride is calculated as the difference between the two recently seen values.

Control dependencies also introduce major performance challenges to superscalar out-of-order processors. Since the efficiency of such processors highly depends on their ability to have the needed supply chain of instructions for parallel execution, the demand for high-bandwidth undisrupted instructions flow is crucial. Branch prediction, in this case, plays a key role not only in reducing the branch penalty but also in helping the processor to fetch instruction across the branch boundaries (using the BTB prediction) and potentially increasing the effective supply chain of candidate instructions for parallel execution. Moreover, the need for highly accurate branch predictor is an essential requirement for such processor. Let's consider a processor with a depth of 100 instructions (instruction window), and let's assume that 20% of instructions in the program are branches. Assume that the branch predictor accuracy is 95%. On average there will be $0.20 \times 100 = 20$ branch instructions in pipeline. The likelihood that all instructions were predicted correctly is 0.9520 which is approximately 36%. A 1% accuracy improvement in the BTB from 95% to 96% will increase the probability from 36% to 44%. This demonstrates the importance of branch prediction accuracy to provide undisrupted fuel of instructions to the out-of-order superscalar processor.

The Reorder Buffer (RoB) is the mechanism that concludes the out-of-order superscalar processor overview. The RoB, which is usually implemented as a cyclic buffer, maintains a record per every instruction being processed. The RoB records are ordered in accordance with the original order of instructions within the program. The RoB is the key mechanism which is responsible for instructions to retire and commit their architectural changes (writes to memory or register file) in order. The commit rule used by the RoB is that an instruction can commit only if it

completed execution and all prior instructions completed successfully. The in-order commit process performed by the RoB is essential to assure the correct order of interrupts (precise interrupts) and to allow speculatively executed instructions (due to branch prediction or value prediction) to wait in the RoB until they can be committed. The RoB also facilitates the flash process in case of branch misprediction since instructions are ordered in the RoB in accordance with program order. This simplifies the detection of the location of flushed instructions since the RoB maintains the needed details on the processing stage of every pending instruction. Last, the RoB guarantees that all architectural state changes will be reflected to the external world as if the program was executed sequentially.

Conclusions

The era of modern computing introduces major challenges accompanied by great opportunities for microprocessors and computing systems. New applications introduce a wide spectrum of requirements with diverse power, performance, and cost constraints. For example, real-time systems present a high bar for real-time requirements accompanied by high performance. Traditional techniques, such as speculation, caching, and prefetching, which were successfully used to speed up performance may be limited for such applications which require deterministic processing time. Another example is related to security which is highly emphasized in a broad range of applications. The emergence of IoT, edge-connected devices, and cloud computing have further intensified processors' vulnerability for malicious cyberattacks. Artificial intelligence and machine learning applications are rapidly evolving and changing the shape of computing by introducing intensive demand for high memory bandwidth and tensor-based processing. In conjunction with these challenges, VLSI process nodes rapidly approach the atomic scale and introduce major design challenges to processor microarchitecture. Microarchitecture implementations have been able to leverage for many decades continuous frequency scale offered by advanced process nodes; however as approaching the physical barriers of frequency, performance becomes highly dependent on processor microarchitecture. In addition to performance challenges, new reliability concerns have become crucial factors which require attention not only at the physical design level but also at the microarchitectural level. The combination of these challenges will sculpt the face of future computing and will require innovative architectural ideas. In this chapter the author reviewed processor microarchitectures starting from single-cycle processor, moving to pipelined processors and finally deep dive to multiple-issue processors. The author presented microarchitectural design considerations, performance matrices, and the limitation on instruction-level parallelism. Acquiring the fundamental microarchitectural knowledge is crucial in order to comprehend future challenges and bring innovative solutions.

Given these challenges and opportunities, it is quite challenging to forecast the evolution of future microprocessor roadmap. Undoubtedly, the VLSI technology will be a major contributor to the shape of the future microprocessors.

In particular, when advanced process nodes are reaching the atomic scale and processor die size continuously grow, the introduction of new 2.5D and 3D integration technologies can offer significant opportunities for further scaling and system integration. Advanced technologies, such as Chip-on-Wafer-on-Silicon (CoWoS), offer advanced multi-die integration with high-bandwidth memory (HBM) on a silicon interposer. Today, 3D integration technologies are already employed by different applications such as routers, FPGAs, and GPUs; however, they have not significantly emerged into traditional general-purpose microprocessor designs. Will microprocessors be able to leverage such technologies and offer ultra large-scale integration of 1000s of cores and peripherals? Such a path may offer tremendous opportunities for performance scale, memory bandwidth enhancements; however, it binds the system to a predefined architectural topology which cannot be controlled by users.

Another force which is changing the paradigm of traditional computer systems is the ongoing shift from control flow-based computing to dataflow-based computing. This important trend, driven today by various applications such as machine learning, high-performance computing, and cryptography, emphasizes microprocessor built-in conflicts. While microprocessors need to provide pretty decent performance for a broad range of applications with diverse requirements, they lack the ability to outperform in specific domains which have different processing requirements. This gap has boosted accelerator development which are tailored for specific applications such as GPUs, TPUs, FPGAs, and ASIC devices. So far, microprocessors have not been able to offer a competitive performance with respect to accelerators. SoCs have integrated various co-processor engines which can be programmed by the CPU to execute specific workloads. Such a solution may introduce a limited performance improvement, in particular when a massive scale of processing is involved. It will be highly interesting to see how microprocessors and accelerators will emerge in the next decades. Will they continue to coexist or will there be a fusion between the two domains? Emerging technologies such as embedded FPGAs offer various opportunities for microprocessors to integrate programmable logic into the processor silicon die. When such technologies become more mature and introduce higher scale and performance, they may offer CPUs a competitive enhancement in the era of accelerated computing.

Finally, powerful emerging memory technologies also introduce a major impact on future microprocessor roadmap. Stack of high-bandwidth memories (HBMs) offers low latency and high-bandwidth integration with processors. Additional approaches such as near- or in-memory computing introduce major memory bandwidth and computational advantages; however, such technologies need to become mature enough before they can be physically integrated into commercial processors. Memristor technologies have also made a major progress in the past decades offering high-density, low power, and resiliency for memory and storage devices. Memristors have not mature yet for commercial product deployment; however, they may significantly change microprocessors' future memory systems.

In summary, future microprocessors incorporate major exciting opportunities that combine emerging technologies which will no doubt change the traditional

paradigm of computing. These changes are driven not only by new technologies but also by the incredible number of new applications of computer systems which affect every field in the day-to-day life.

References

Alpha 21264 Microprocessor Data Sheet (1999) Revision 1.0, Feb 1999. Compaq Computer Corporation

Gabbay F, Mendelson A (1997) Speculative execution based on value prediction. Technical report, Technion

Gabbay F, Mendelson A (1998) Using value prediction to increase the power of speculative execution hardware. ACM Trans Comput Syst 16(3):234–270

Hennessy JL, Patterson DA (2011) Computer architecture. A quantitative approach, 5th edn. Morgan Kaufmann Publishers Inc., San Francisco

Kane G (1988) MIPS RISC architecture. Prentice Hall, Inc.

Lee JKF, Smith AJ (1984) Branch prediction strategies and branch target buffer design. IEEE Comput Mag 17(1):6–22

Lipasti MH, Shen JP (1996) Exceeding the dataflow limit via value prediction. In: Proceeding of the 29th annual ACM/IEEE international symposium on microarchitecture, pp 226–237

McFarling S (1993) Combining branch predictors. Digital Western Research Laboratory, Technical Report

Mittal S (2019) A survey of techniques for dynamic branch prediction. Concurr Comput Pract Exp 31(1) p. e4666

Patterson D, Watterman A (2017) The RISC-V reader: an open architecture atlas. ISBN-13: 978-0999249116. Strawberry Canyon Publishers

Simcha Gochman RR et al (2003) The Intel Pentium M Processor: microarchitecture and performance. Intel Technol J

Skadron K, Ahuja PS, Martonosi M, Clark DW (1998) Improving prediction for procedure returns with return-address-stack repair mechanisms. In: Proceedings. 31st annual ACM/IEEE international symposium on microarchitecture, Dallas, pp 259–271. https://doi.org/10.1109/MICRO.1998.742787

Sloss AN, Symes D, Wright C (2004) Chapter 2 – Arm processor fundamentals. In: The Morgan Kaufmann series in computer architecture and design, ARM system developer's guide. Morgan Kaufmann, pp 18–44. ISSN 15459888, ISBN 9781558608740. https://doi.org/10.1016/B978-155860874-0/50003-4

von Neumann J (1945) First Draft of a Report on the EDVAC, archived from the original (PDF) on 14 Mar 2013, retrieved 24 Aug 2011

Wang K, Franklin M (1999) Highly accurate data value prediction using hybrid predictors. In: Proceeding of the 30th annual ACM/IEEE international symposium on microarchitecture, pp 281–290

Yeh T-Y, Patt Y (1991) Two-level adaptive training branch prediction. In: Proceedings of the 24th annual international symposium on microarchitecture, pp 51–61

The Architecture

2

Avi Mendelson

Contents

Introduction . . . 48
Terms and Notations . . . 50
Laws and Models in Microprocessor/System-on-Chip (SoC) Architectures . . . 50
ISA Selection and Considerations . . . 54
CISC: Complex Instruction Set Computer . . . 54
RISC: Reduced Instruction Set Computer . . . 61
The RISC-V Approach for ISA . . . 69
Summary of ISA Selection . . . 73
Vector and SIMD Extensions . . . 73
Cross-Layers Optimizations . . . 79
Background . . . 80
Delayed Branch in MIPS . . . 80
The User-Defined Microcode Programming . . . 81
VLIW Architectures . . . 81
HW/SW Codesign: The CUDA Approach . . . 82
ISA Agnostic Systems . . . 84
The Use of Intermediate Representations . . . 84
Binary Translation . . . 85
Summary . . . 86
References . . . 86

Abstract

This chapter focuses on the architectural aspects of single-core architecture. It starts with a discussion on the different design philosophies of choosing the processor's instruction set architecture and continues with a discussion

A. Mendelson (✉)
CS Department, Technion, Haifa, Israel
e-mail: avi.mendelson@technion.ac.il

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_1

on optimizations that break the barriers between the traditional software and hardware interfaces. Finally, it discusses the main differences between general-purpose processors and dedicated (domain-specific) architectures.

Keywords

RISC · CISC · Domain-specific architecture · VLIW · SoC (System on a chip)

Introduction

Abstraction is a fundamental concept in computer systems. In (Patt and Patel 2003), the authors present nine layers of abstractions (see Fig. 1) that assemble every computer system. Such a partition simplifies each layer and develops multiple alternatives that differ by their optimization points, for example, low power, best performance, fast time to market, etc. It also helps to improve the interfaces between the software perspective of the system and the actual implementation.

This chapter focuses on the architecture of a single-core processor. Special attention is given to the design considerations when choosing the Instruction-Set-Architecture (ISA) since this layer interfaces between the system's software and hardware perspectives. Please note that these perspectives do not always agree. For example, in some cases, the microarchitecture view of the ISA is different than the software's view.

Most current ISAs are based on a Von Neumann model (Neumann 1945). The model assumes that the system is built out of three main components: memory, execution unit, and I/O devices that can also interface with the "external world," as depicted in Fig. 2.

Figure 2 shows the three main components of the model; please note that although the I/O subsystem is shown as having a separate hardware implementation, most implementations share the same hardware components between these two

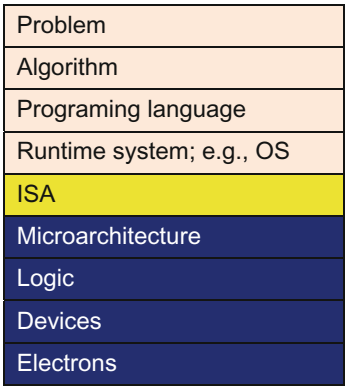


Fig. 1 Layers of abstractions

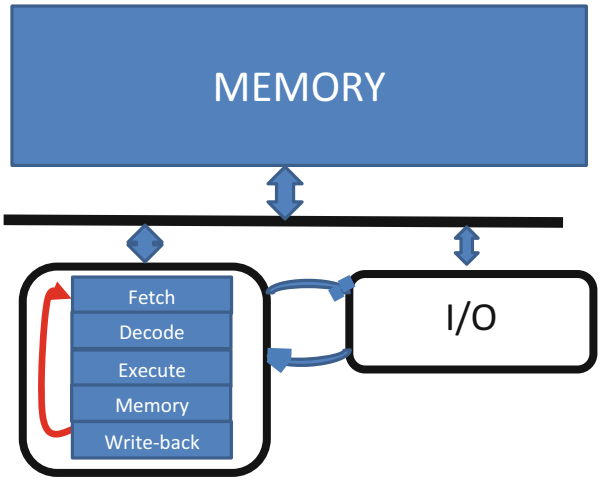


Fig. 2 Von Neumann model of execution

modes of operations. A dedicated bit in the status register indicates the current execution state, meaning it is used to help distinguishing between these modes. Thus, the same hardware behaves differently when the system is in execution mode and when the system is in I/O mode (supervision mode).

The Von Neumann model does not consider the cache hierarchies since it is transparent to the software model and mainly aims to improve the processor's performance. Thus, it is usually considered as part of the microarchitecture (and will be discussed in the next chapter). Registers, on the other hand, although considered to be part of the memory hierarchy, do have a significant impact on software, and so, they are considered to be part of the interface between the software and the hardware, and so assumed to be part of the ISA.

Figure 2 also depicts the different execution stages of the Von Neumann. It shows a simple in-order five-stage pipeline architecture where (1) instructions are fetched from memory with respect to a special register called program counter (some architectures call it instruction counter), (2) the interpretation of the instruction, that is, what the system needs to perform, is determined at the decode stage, (3) the execution stage is dedicated to performing the calculation and to perform address calculations, (4) during the memory stage, the system uses the address that was calculated in the third stage to read data or to write data to memory, and (5) finally at the write back (commit) stage data gets exposed to the external world; for example, written back to registers.

There are many different types of ISAs, such as CISC (complex instruction set), RISC (reduced instruction set), vector operations, and mix/hybrid modes. This chapter attempts to explain why multiple ISAs were created, and what is the cost of using a "wrong" ISA. We will start this chapter with a discussion on some of the "traditional" classes of ISA and extend the discussion to methods, such as

using intermediate representations (IR) and binary translations to overcome these limitations.

Before diving into the details, let us review some of the terms and notation we will use.

Terms and Notations

- ISA – Instruction Set Architecture – represents the compiler and other software packages view of the hardware.
- ILP – instruction level parallelism; how many instructions the system can execute every cycle.
- CPI – Cycles Per Instructure $= 1/\text{IPC} = \frac{\text{Total number of cycles required to execute the program}}{\text{Total number of instructions executed in the program}}$
- Performance: It is usually measured as the time it takes to compute a specific task. One can estimate performance as Performance $=$ IC $\times$ CPI $\times$ Clock cycle
- Amdahl law (Amdahl 1967, 2013; Gustafson, Reevaluating Amdahl's law, 1988)

$$t' = t * \left[(1 - F) + \frac{F}{S} \right] \tag{1}$$

The new execution time $=$ the fraction that was not affected $+$ the optimized portion. This law is used in computer architecture to indicate that it is recommended to optimize code sections that are often be used.

- Memory footprint: The code and data size needed to keep a program's complete instructions and data.
- Power: The amount of electricity a device is consuming at any point in time.
- Energy: The amount of electricity a device consumes during a period of time or when executing a task.
- Hardware complexity: In this chapter, we assume that hardware complexity is proportional to the size of the silicon that it takes to implement it.
- Backward compatibility: SW that was compiled to run on an older version of the hardware will also run on a new generation of hardware. Please note that some systems require that the new generation of hardware will run the old code, at least with the same performance (or better performance) as it was used to be executed using the previous generation of that hardware.
- Load/Store architecture: Architecture that all mathematical operations are done between registers, and data is always being fetched to a register via Load instruction and written back to memory using Store instructions (Fig. 3).

Laws and Models in Microprocessor/System-on-Chip (SoC) Architectures

It is imperative for any domain of knowledge to be strictly governed and guided by a set of fundamental rules (This subsection was contributed by Prof Anupam

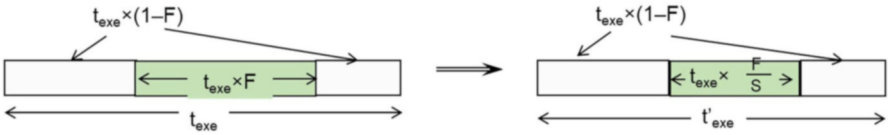


Fig. 3 Amdahl's law

Chattopadhyay ()), be it in the form of laws, models, or even hypotheses. Computer architectures are no exception. In the following, we review some of the most important theoretical underpinnings of computer architecture, which serve as the guiding principles of generations of architecture design.

The laws and models are divided in three abstractions, namely, *technology*, *architecture*, and *application-specific focus*.

Technology-level laws did originate from the underlying semiconductor technology upon which modern computer architectures are built. These laws had an overarching influence on architectural design as briefly explained in the following.

- Moore's Law, coined by Gordon Moore, gave the prediction that the number of transistors in an integrated circuit doubles every 2 years. This became a guiding principle for generations of semiconductor technology and resulted in boosting the performance of computer architecture. However, there is a noticeable slowdown in the progress in leveraging Moore's law since 2010s, due to numerous effects across the complete architecture design stack (e.g., leakage power, reliability, manufacturing defects, memory wall, limits of parallelism). As a result, architecture designers are moving to alternative avenues to boost efficiency, such as coarse/fine-grained parallelism (multicore platforms, reconfigurable architectures), in-memory computing, photonic computing, and superconducting technologies.
- Dennard's Power Scaling (Dennard et al. 1974) stated that with increasing transistor density in an integrated circuit, there is a proportional decrease in capacitance and voltage – resulting in the same power consumption. This indicated that the performance per watt increases (doubling every 2 years) since, with more transistors, higher performance can be extracted at architecture level. This scaling phenomenon coexisted together with Moore's Law from the mid-1970s until it was shown to be broken due to the effects of leakage current and resulting thermal runaway.

Architecture-Level Laws and Models

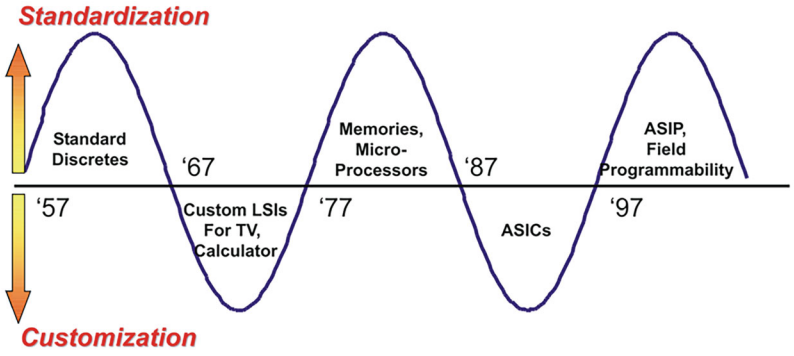
- Turing Machine (Turing 1938) is the fundamental model of any computing machine. Proposed by Alan Turing in 1937, it showed that through a basic model of program, data, and a computing pinhead, it is possible to capture the steps of anything that is computable. Subsequently, it became the fundamental abstraction for modeling computers and the basis for analyzing computational complexity of any given algorithm. Turing machine also represents the most general form

of computing abstraction, which includes other abstractions, such as finite state machines, and combinational logic.

- Bulk-Synchronous Parallel (BSP) (Valiant 1990) Model is one of the architectural abstractions to capture parallel architectures. In an extension of Turing Machine and earlier Parallel Random-Access Machine (PRAM), BSP model was specifically designed to capture the overheads of communication and synchronization in a parallel architecture. Several distributed computing programming flows use BSP as the underlying technology, such as Apache Hama and MapReduce.
- Flynn's Taxonomy (Flynn 1972) presented a formal classification of computer architectures based on instruction and data streaming. It has four classes: Single-Instruction Single Data (SISD), Single-Instruction Multiple Data (SIMD), Multiple-Instruction Single Data (MISD), and Multiple-Instruction Multiple Data (MIMD). Any computing architecture can be categorized as per this taxonomy.
- Amdahl's Law (Amdahl 1967) states that for any given application, the overall achievable speedup is essentially limited by the inherently sequential part of the application. This shows that even by using highly parallel underlying architectures, one may not be able to obtain matching speedup. In a sense, this is a pessimistic view of architecture evolution, which was countered by Gustafson's Law. Amdahl's Law was also revisited from the perspective of multicore architectures (Hill and Marty 2008), which shows that it is profitable to have asymmetric (heterogeneous) and dynamic (reconfigurable) structures for multicore architectures.
- Gustafson's Law (Gustafson, Reevaluating Amdahl's law, 1988) essentially revisited and addressed the shortcoming of Amdahl's law. Instead of being restricted by the sequential part, this law states that it is possible for a programmer to keep on expanding the parallel tasks so as to completely leverage the underlying architectural parallelism. This led to the ideas of embarrassingly parallel applications and eventually coarse-grained parallel architectures like graphics processing units.
- Bitlet Model (Ronen et al. 2021), in recent times, looks into the possibility of using memory as a computing model. Traditionally, memory objects were purely used as a storage. However, with recent advances in memory controller architectures and purely in-memory computing devices, it was only natural to take the advantage of these technologies to the growing crisis of memory/bandwidth bottleneck in highly data-intensive applications. Bitlet model proposed an analytical way to compare the efficiency of running the same kernel inside memory or in the processing unit. Consequently, it was shown also through practical demonstrations of emerging non-volatile memories (RRAM, STTRAM) that, for given sets of applications, such as image processing, machine learning, and genomic sequencing, it might be significantly advantageous to perform in-memory computing.

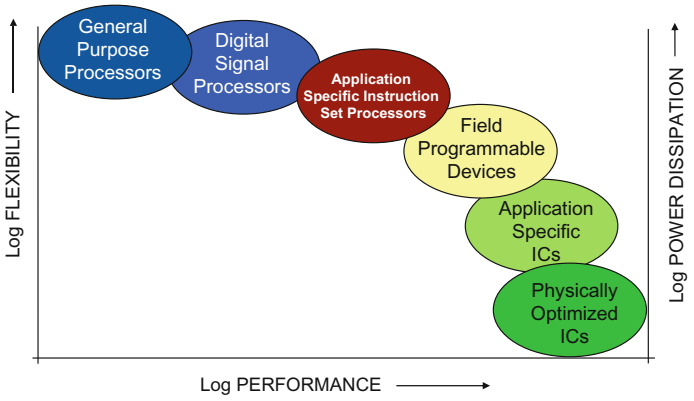
Application-Specific Focus

- Makimoto's wave (Engineering, Makimoto's Wave, 1991) showed that the design community periodically swings between customization and standardization of the system component. With the advent of each new architectural innovation, there is a significant push towards customization (in turn, increasing the application performance), which after a while reduces to standardized, and less flexible, designs catering to a wider segment of applications. The move from customization to standardization is also correlated with the growth of trained engineering manpower and robust, sophisticated design automation flows.



Source: Electronics Weekly, Jan. 1991

- Flexibility-efficiency trade-off between various architectural choices can be nicely captured following different performance metrics. These components are collectively put in a given SoC for developing the heterogeneous/asymmetric multicore SoC, as advocated in the work of Hill and Marty (Amdahl's Law in the Multicore Era).



ISA Selection and Considerations

Different processors aim at various markets; some target multiple markets, termed "general purpose"; others target specific markets such as IoT devices or sensors. The market dictates the optimization points and the design philosophy of each processor. Here are a few examples:

- Intel processors embraced the "backward compatibility" of code.
- ARM is focusing at low power design (Engineering, Makimoto's Wave).
- CUDA cores target embarrassingly parallel code.

This chapter examines three types of ISA: CISC, traditional RISC, and modern RISC; for example, RISC-V. For each type, we describe its optimization points, provide a few examples of processors using it, and discuss the pros and cons of the approach.

CISC: Complex Instruction Set Computer

A complex instruction set computer aims to make each instruction as expressive as possible so that a compiler can use a minimum number of instructions when translating a high-level language into assembly code. The motivation was to reduce the size and complexity of the software and support backward compatibility since more instructions could always be added. Most of the CISC architectures also add sophisticated addressing modes to increase the expressiveness of each assembly instruction. CISC architectures usually use a variable instruction length and short decoding for instructions that are frequently used and longer instruction length for instructions that are rarely used.

Vax machines (Macro 1996) used to be a leading CICS architecture that brought many excellent ideas to the market. It had assembly-level instructions such as adding an element to a double-link list which used to be a very common operation in VMS operation systems. Thus, VAX architecture does not exist anymore, so we choose to examine the evaluation of the Intel family of architectures (some of them are listed in Table 1) to demonstrate how the use of CISC architectures helps to maintain the notion of backward compatibility.

The following subsection starts with evaluating the ISA of early Intel generations, namely, 8088 and 8086. Then we extend the discussion to examine the impact of extending the architecture to 32 bits, and we conclude this subsection with a discussion on the current X86-64 architectures.

The Baseline: Looking at the ISA of the 8088 and the 8086 Processors

The 8088 processor was one of the first processors Intel launched in 1979 (Singh 1988). It used an 8 bits data path and 20 bits address width. The architecture was based on16 bits registers (see Fig. 4), adopted the Von Neumann principles of operation, and implemented a simple in-order architecture. The system uses physical

Table 1 Evaluation of Intel processors

	Intro		Address space			Few features
			Linear	Virtual	Physical	
X86	1978	8088, 8086, 80,186	16	N/A	20	
	1982	80,286	16	30	24	
	1985	386,486, Pentium	32	46	32	32 bit ISA, Paging, MMX
	1995	Pentium Pro, Pentium II,III	32	46	36 (PAE)	
	2000	Pentium4, Pentium M Dual Core	32	46	36	Hyperthreading, XD bit
X86-64	2004	Pentium 4 (Prescott)	36			EM64T
	2006	Core 2	36			SSE3
	2008	I7, I5 (2009), I3 (2910)	40			EPT (virtualization) SSE4.2
	2015	Skylake, Kaby Lake, coffee Lake	46			AVX512
	2018-today	ICElake, Tiger Lake, Alder Lake	57			Dual ring, neural accelerator

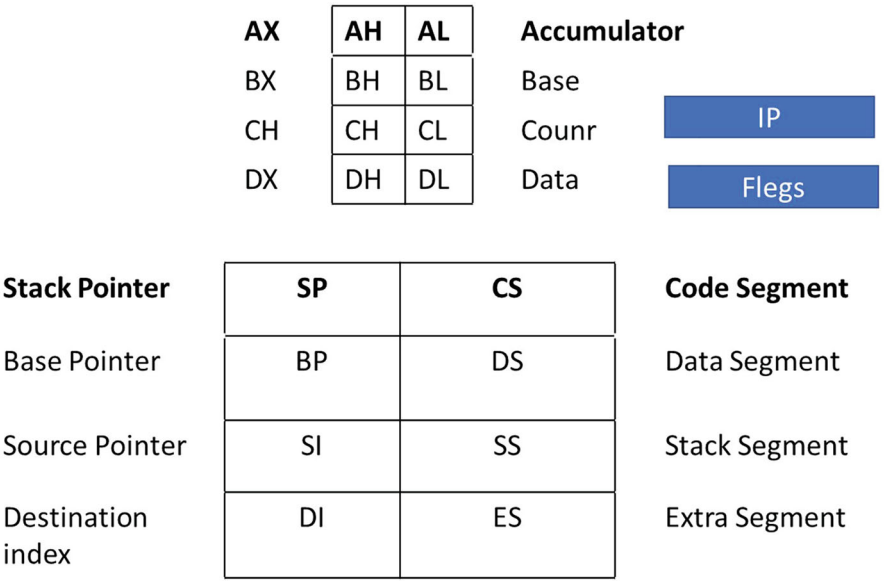


Fig. 4 Register file of 8088 processors

memory only and divides it into four segments. Thus, four segment registers were used to point to the correct memory region. The Code, Data, and Stack segment registers were automatically selected with respect to the type of operation the system performed. At the same time, the ES segment register requires an explicit notation since it was used for sharing data among different tasks.

Since registers were 8 or 16 bits long, a Load or Store instruction could access only a window of 216 = 64 Kbytes, but the maximum allowed physical memory size was limited to a megabyte (20 bits). A task running on that processor could access up to four segments, 64 KB each, and the rest of the memory could be used via (1) the use of multitask software or (2) manipulation of the segment registers.

The complex instruction set of the X86 architecture allows each instruction to be quite expressive; Table 2 lists some of the mathematical operations it support (please note that 8088 and 8086 did not have direct support for floating-point operations).

Table 3 shows all the different addressing modes of an ADD instruction, followed by the impact of the addressing modes on the execution time of the operation.

So far, we have focused on a subset of a few basic instructions of the X86 architecture, but the "basic" X86 ISA contains more than 100 instructions; each can be 1–6 bytes long, and the execution time could vary from a single cycle to hundreds of clock cycles each.

Table 2 Math operations in 8086/8088

Add		SUB	MUL			DIV	
ADD	Add byte/word	Sub.	Subtract	MUL.	Multiply	DIV	Divide
ADC.	Add with carry	SBB.	Subtract + borrow	IMUP	Integer MUL	IDIV	Integer divide byte or word
INC.	+1	DEC.	−1	AAM	MUL in ASCII	AAD	ASCII adjusts for division
AAA.	Add in ASCII	NEG	Negate				
DAA.	Add in decimal	CMP.	Compare			MISC	
		AAS	SUB-ASCII			CBW	Convert byte to word
		DAS.	Sub-decimal			CWD	Convert word to doubleword

Table 3 Address modes of ADD instruction (X86)

ADD – Add two numbers			
Operands	Clock	Decode bytes	Coding example
Reg, Reg	3	2	ADD AX, BX.
Ref, Memory	9 + EA	2–4	ADD DI, ALPHA
Memory, Reg	16 + EA	2–4	ADD. Beta, AX
Register, immediate	4	3–4	ADD. BX,2
Memory, immediate	17 + EA	3–6	ADD. Alpha,2
Accumulator, immediate	4	2–3	ADD AX,200

As a result, the code generated by the compiler consumed a relatively small area (address space) since each instruction was quite expressive. It helped to reduce the communication needs between the memory and the execution units, but the overall implementation of the system (hardware) was quite complex. As a result, the operation clock was relatively slow, and many bugs were discovered during the product's lifetime.

IA32 Architectures

When Intel expanded its architecture to 32 bits, it also added quite a few technologies, such as

- ISA was extended to support floating point operations (FP).
- Vector operations, in the form of MMX, were introduced.
- Pages were introduced in the form of "memory flat model" and were added on top of the segmentation.
- Physical address space was extended to support 32 bits (4 gigabyte).
- Protection levels were added.
- Many more features.

Figure 5 depicts the internal state of the IA32 core. CISC architecture allows the companies (e.g., Intel and AMD) to extend and adjust to new ISA in a relatively straightforward manner since existing code could continue running using the new generation of the core, with a performance that was, in most cases, at least the same as before. But using the same decoding scheme to preserve compatibility caused some performance penalties since the efficient and short decode schemes were already being used by instructions that were less frequently used by the new applications. On the other hand, the development time of a new generation of processors was significantly reduced since the designers could reuse significant subsystems of the old generation as part of the new design which shortened the overall development time.

Extending the Architecture to 64 Bits (X86-64 ISA)

Intel made different attempts to extend the X86 ISA into 64-bit architectures. The first attempt was suggested by the Itanium line of processors. This generation of processors was incompatible with the X86 ISA (Sharangpani 1999a, b). Unfortunately, this line of products failed to be a commercial success; Intel stopped producing them, so we will not extend the discussion on that.

The compatible extension of X86 ISA to 64 bits was developed by AMD (AMD 2000) and was adopted by the community. Although this extension's primary goal was to access very large physical and virtual address spaces, the byproduct was the introduction of direct support for many new technologies, such as operations of wide vectors, direct access to memories over PCI buses, and more.

The new extension also caused some of the old technologies to become redundant, for example, the use of segmentation. Thus, under X86-64, segment registers can be ignored. But in order to allow compatibility, the segment registers: CS, DS, ES, SS are treated as if each segment base is 0.

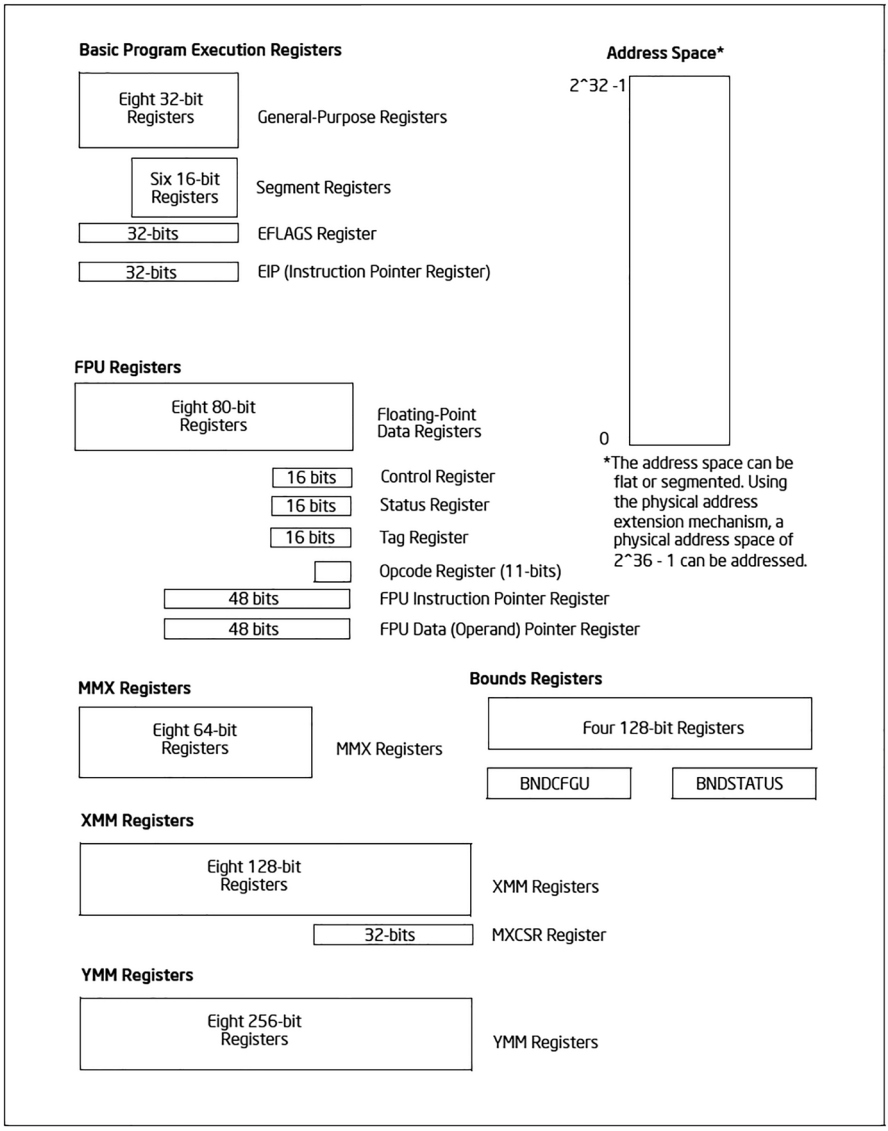


Fig. 5 IA-32 Basic execution environment (32 bits)

IA-64 Registers

In order to support compatibility, the IA-64 supports all different lengths of registers (Fig. 6):

- 64-bit general-purpose registers (RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP, or R8-R15)

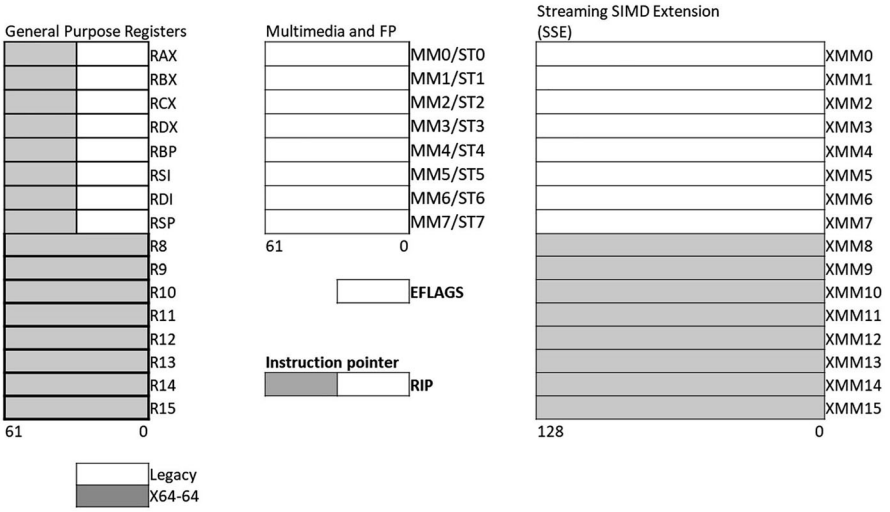


Fig. 6 Registers in IA-64 architectures

- 32-bit general-purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP, or R8D-R15D)
- 16-bit general-purpose registers (AX, BX, CX, DX, SI, DI, SP, BP, or R8W-R15W)
- 8-bit general-purpose registers: AL, BL, CL, DL, SIL, DIL, SPL, BPL, and R8B-R15B are available using REX
- MMX registers (MM0 through MM7)
- XMM registers (XMM0 through XMM15) and the MXCSR register
- Control registers (CR0, CR2, CR3, CR4, and CR8) and system table pointer registers (GDTR, LDTR, IDTR, and task register)
- Debug registers (DR0, DR1, DR2, DR3, DR6, and DR7)
- MSR registers
- RDX:RAX register pair representing a 128-bit operand

Please note that some of the registers are mapped onto the same area (see Fig. 6) to save power, area, and time when the system needs to store or restore the content of a process.

Adjusting the Architecture to Support New Technologies

Extending the architecture with new instructions provides efficient support for new technologies. Thus, almost any new generation of Intel cores comes with specific extensions. For example, a quick look at the Intel® 64 and IA-32 Architectures Manual (Intel 2021a, b) presents the following technologies:

- Advanced Matrix Extensions (Intel® A.M.X.)
- ENQCMD/ENQCMDS instructions and virtualization support

- Intel® TSX Suspend Load Address Tracking
- Hypervisor-managed Linear Address Translation
- Architectural Last Branch Records (LBRs)
- Non-write-back lock disables architecture
- Bus lock and VM notify features
- Resource Director Technology feature updates
- User interrupts
- Updates on the usage of performance monitoring
- Enhanced hardware feedback interface (EHFI)
- Linear Address Masking (LAM)
- Machine error codes for processors based on Sapphire Rapids microarchitecture
- IPI Virtualization

The above list is only partial, and many more are proposed, such as SGX – Security Guarded Extension was proposed as an extension to allow the creation and protection of enclaves; secure memory regions, ISA support for machine learning algorithms, and more

Some of these technologies require the introduction of new data types

(a) Traditional data types

(b) New Data Type (2021)

Recent X86–64 architectures add support for new short floating-point formats to provide efficient support for machine learning algorithms, Table 4. We may assume that the next generation of Intel processors will extend the notion of vectors to the new notion of multidimensional matrix operations.

Table 4 Data types for Intel X86-86

	Char	Integer	Floating point	Vector
Byte – 8 bits	Yes	Short		INT8
Word – 16 bits	Yes	Yes	Yes	Short
Doubleword – 32 bits		Yes	Sing precision	Yes
Quadword – 64 bits		Yes	Double precision	MMX (XMM.)
Double quad – 128 bits		Packed integer	Packed Float	AVX (YMM.)
Vector 512 bits				AVX-512 (ZMM.)

Summary

CISC architectures, in general, and Intel architectures in particular, were developed under the assumption that software compatibility is the most crucial feature the architecture needs to preserve, even at the cost of extra complexity, since it can significantly help to achieve better performance and to provide significant benefits to the end user. This approach enables an easy feature migration path and naturally supports backward compatibility. However, the CISC architecture presents a few inherent challenges:

- The critical path determines frequency, and accelerating all stages of a complex system is costly.
- Suboptimal decoding scheme.
- The architecture is error-prone.
- Difficult to decode multiple instructions in parallel.
- Advanced addressing modes, such as supporting arithmetic operations between memory and registers, make optimizations very difficult.
- Power is a major issue in modern design, and complexity affects power consumption.

Intel solves (or at least eases) many of these issues at the microarchitecture level; all Intel architectures, starting at P6, use out-of-order execution and micro-operation as the internal assembly code of the machine.

We will extend the discussion on how microarchitecture helps in the next chapter.

RISC: Reduced Instruction Set Computer

Unlike CISC architectures that aim to make each instruction as expressive as possible, the RISC design philosophy calls for "simplicity"; this approach was motivated by the following observations:

- Amdahl (Amdahl 2013) low calls for accelerating the part we are using the most (see Eq. (1))
- At runtime, only a handful of instructions are frequently executed (Hennessy and Patterson 2007; Patterson et al. 1979)
- Simple design allows an increase in frequency
- Load/Store architecture allows
  - To increase the number of registers. In return, it will reduce the number of overall memory operations considered the main bottleneck of computer systems.
  - Using three operands per instruction enables many optimizations that improve the execution time.
  - The large register window enables an efficient exchange of parameters when calling a procedure and returning to the caller.

IBM research was the first to experiment with this new approach, with their experimental processor IBM 801 (Radin 1983). The idea yields a few competing research projects, for example, the SPARC processor (Garner 1988), the RISC-I processor, and the MIPS architecture (Kane and Heinrich, MIPS RISC architectures, 1992.). We start this chapter with a discussion on SPARC and MIPS architecture before describing the ARM family of RISC processors (Furber 2000).

MIPS

The MIPS architecture started as an academic project at Stanford University (Hennessy et al. 1982), and soon after, a company was established to commercialize it (Kane and Heinrich, MIPS RISC architectures., 1992). The processor had different generations, but in this chapter, we mainly focus on the first generation, termed MIPS-I, since most of the concepts of the entire family already appear there.

MIPS-I processor was a Load/Store architecture that presented a simple architecture to increase the frequency (concerning CISC at that time) and allowed the creation of many new compiler optimizations. The MIPS processor has 32 registers, 32 bits each; some have a specific goal; for example, R0 is hardwired to zero, and register R31 is used as a link register. The machine assumes in-order, Von-Naumann architecture but allows multiplication and division instructions to be executed asynchronously as long as dependencies are maintained.

The program counter is 32 bits long, but the lower 2 bits are wired to "0" since all MIPS-I instructions are 4 bytes long and aligned to word boundaries. Simplicity was the main target of all RISC architectures, including MIPS, so instructions are always 4 bytes long and have one of the following three internal formats; R (Register), I (Immediate), and J (Jump), as depicted in Fig. 7, below:

To improve performance, MIPS introduced the notion of a branch delay slot that provides the compiler the ability to help the microarchitecture. We will extend the discussion on that in section "RISC: Reduced Instruction Set Computer".

MIPS-I was the first RISC architecture to allow complex instructions, such as floating-point operations, to use multiple cycles to complete an operation. To support FP operations, MIPS added 32 floating-point registers that could also be used as 16 double-precision FP registers. Adjust registers could also be paired in order to support double-precision numbers and double-precision arithmetic operations.

Type	Format					
R	Opcode (6)	RS (5bits)	RT (5bits)	RD (5bits)	Shamt(5)	Func(6)
I	Opcode (6)	RS (5bits)	RT (5bits)	Immediate(16 bits)		
J	Opcode (6)	Address (26 bits)				

Fig. 7 Instruction formats of MIPS-I

SPARC ISA

SPARC (Garner 1988) is another RISC core originally developed in Berkeley and supported by Sun. It also had a simple instruction format (although more complicated than MIPS), as shown in Fig. 8. Although SPARC was affected by other RISC architectures, it introduced quite a few unique features, such as a new use of the register window, the support of coprocessor, and more.

Sparc presents a new use of a Register-Windows; a "scratchpad" of fast memory region (SRAM), being used as a cyclic buffer of registers. In SPARC, the register window could contain 40–520 registers, organized as eight global registers and between 2 and 32 overlapping register banks. At any given time, a process can access the eight global registers and a dedicated 32 registers managed as a sliding window out of the scratchpad.

To manage the register window, SPARC added a special register, the CWP, that was also added to the process's context (meaning it was saved and restored at during context switch). The CWP points to a continuous area in the scratchpad and is served as the currently active register region of a process (see Fig. 9). The active register window was divided into three subregions; eight registers were treated as "IN" registers, eight registers as "Local" registers, and the last set of eight registers as "OUT" registers. When a caller calls a procedure (which can be itself in the case of a recursive call), the sliding window was pushed down so that the new CWP points to the region that used to be the OUT region before (now it is considered as IN) and new memory region is allocated within the scratchpad for Local and

	0-1	2-6	7-9	10-12	13-17	18	19-26	27-31
Format 1 call	OP (2)	Displacement (30)						
Format 2 SETHI	OP (2)	RD(5)	OP(3)	Immediate (22)				
Format 2	OP (2)	a CC(4)	OP(3)	Immediate (22)				
Format 3 i=0	OP (2)	RD(5)	OP(6)		RS1(5)	i	FP-OP(7)	RS2(5)
Format 3 i=1	OP (2)	RD(5)	OP(6)		RS1(5)	i	Immediate (13)	

Fig. 8 SPARC instruction types

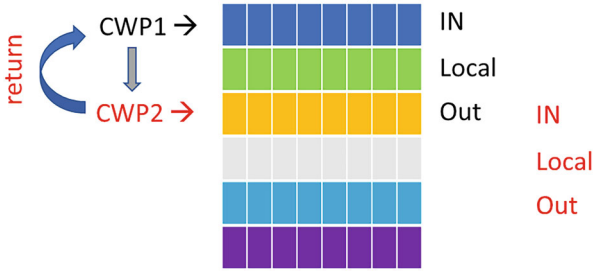


Fig. 9 Register window in SPARC

Table 5 SPAC coprocessor instructions

LDC	Load Coprocessor	Moves data from memory into a coprocessor register
LDDC	Load Double Coprocessor	Moves a word from memory into the coprocessor state register
LDCSR	Load coprocessor state register	
S.T.C., STDC, LDCSR	Store from CP to memory	
CBccc	Branch on coprocessor condition codes	Branch that depends on the conditional code of the coprocessor
CPop	Coprocessor operate	Implementation-dependent

OUT register regions. The old CWP was restored at return time, and the overlapped register region can be used for returning parameters.

The overlap between the OUT registers of the caller procedure and the IN registers of the callee procedure enables a very efficient mechanism to transfer parameters and allows a fast implementation of a procedure call. The downside of this mechanism is that the hardware does not support automatic protection of the instruction window against overflow, for example, in case of deep recursion. It is up to the process to manage it.

SPARC also presented a new method to support coprocessors. Although coprocessors were optional in SPARC, the ISA allowed to consider them as part of the machine's general pipeline and to manage them accordingly.

The instruction set supports a single, implementation-dependent coprocessor (see Table 5). The coprocessor could have an independent set of registers but must have a state register and a condition register. All coprocessor data and control/status registers could be accessed via two special load/store coprocessor instructions that use one of the formats presented in Table 5.

The ability to integrate accelerators and user-defined instructions as part of an ISA of a RISC architecture helps to extend the RISC model but requires more complicated design, vast support from compilers, libraries, and other software tools.

ARM: Advanced RISC Machines

ARM is the most common ISA today since it has been used by most cellular phones, IoT devices, and a large portion of notepads and servers. ARM ISA, similar to X86, contains thousands of instructions but still uses the RISC principles. It is achieved by having different versions of ISA, not always incompatible with each other, allowing extensions, and implementing some of the ISA as coprocessors. In the following subsections, we will cover these aspects. Table 6 shows the entire instruction set of Thumb-I architecture (ARM 1995). When ARM extends its ISA to 32 bits, it still supports the Thumb ISA (parallel to the ARM-32 bits ISA). Thus, such architectures present a hybrid approach of 32 bits registers and addressing modes but 16 bits-based instruction set.

Using such a hybrid architecture allows achieving a compact code while still taking advantage of the 32 bits registers and 32 bits data path and the ability to support large memory. Under THUMB, the programmer can use eight general

Table 6 Thumb instruction formats

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Move shifted register	0	0	0	Op		Offset5					Rs			Rd		
Add/subtract	0	0	0	1	1	I	OP	Rn/offset3			Rs			Rd		
Move/compare/ add/subtract immediate	0	0	1	Op	Rd			Offset8								
ALU Operations	0	1	0	0	0	0	Op				Rs			Rd		
Hi register operations/ branch exchange	0	1	0	0	0	1	Op		H1	H2	Rs/Hs			Rd/Hd		
PC-relative load	0	1	0	0	1	Rd			Word8							
Load/store with register offset	0	1	0	1	L	B	0	Ro			Rb			Rd		
Load/store sign-extended byte/halfword	0	1	0	1	H	S	1	Ro			Rb			Rd		
Load/store with immediate offset	0	1	1	B	L	Offset5					Rb			Rd		
Load/store halfword	1	0	0	0	L	Offset5					Rb			Rd		
SP-relative load/store	1	0	0	1	L	Rd			Word8							
Load address	1	0	1	0	SP	Rd			Word8							
Add offset of stack pointer	1	0	1	1	0	0	0	0	S	SWord7						
Push/pop registers	1	0	1	1	L	1	0	R	Rlist							
Multiple load/store	1	1	0	0	L	Rb			Rlist							
Conditional branch	1	1	0	1	Cond				Soffset8							
Software Interrupt	1	1	0	1	1	1	1	1	Value8							
Unconditional branch	1	1	1	0	0	Offset11										
Long branch with link	1	1	1	1	H	Offset										

registers, R0-R7, the Program Counter (PC), the stack pointer register (SP), the link register (LR), and the process state control Register CPSR. The system also uses some hidden registers, such as the SPSR that keeps the saved state of the system, to allow fast interrupts; some versions use performance counters and more.

Thumb has two versions, Thumb-I and Thumb-II. Thumb-I mainly represents 2 bytes instruction decoding, and Thumb-II extends it with mainly more 4 bytes instruction decoding so that Thumb-II represents a hybrid instruction decoding length (2 and 4 bytes).

ARM7-32 Bits

ARM7-32 bits represent a family of chips, each aimed at a different market, with different optimization points. Thus, most of the differences appear at the system level (and not at the level of the ISA). The family is divided into mainly three categories (AKA profiles):

- The A profile – targets high-performance open application platforms.
- The R profile – targets high-end embedded systems such as embedded, high-performance real-time systems.
- The M profile – targets microcontroller-based systems.

Since this chapter mainly focuses on the ISA of ARM7, we will ignore the system-level differences among these classes. The format of different instruction formats of ARM7 appears in Table 7.

In order to allow better control over power consumption, ARM7 adds predication bits to control many of its instructions. Predications were proven to be an efficient mechanism to allow the compiler to indicate to the architecture if an instruction needs to be executed or not. When predications are used, the execution of instruction takes place only if the condition bits are reset (0). This technique is widely used in modern parallel architectures such as CUDA.

Please note that the basic ARM7 ISA does not support floating point or SIMD operations. Instead, these operations used to be executed as accelerators, taking advantage of the interface for coprocessors.

The two instruction sets (Thumb and 32 bits) can coexist, so the architecture maps the two architecture states, so they overlap, as depicted in Fig. 10. The ARM architecture does not support a hybrid mode, where instructions from the two sets of ISAs can interchange, but the system can be either at 16-bit mode or 32-bit mode. In order to switch between the modes, a special instruction, BX, was added. If used with state bit (bit 0) set, the system will switch to Thumb (16-bits) mode. Transition to Thumb state will also occur automatically on return from an exception (IRQ, FIQ, UNDEF, ABORT, SWI etc.), assuming the exception happened when the processor was in Thumb state.

AA64 Architecture

The AARCH64 (Pyeatt and Ughetta 2019) is a new ISA that ARM presented to support the new generation of 64-bit architectures. This subsection does not intend to serve as a tutorial on AARCH64 architecture. Still, it mainly discusses the scalability issues of different architectures belonging to the same family of

Table 7 ARM 32 bits instrution set format

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Data Processing/ PSR Transfer	Cond				0	0	I	Opcode				S	Rn				Rd				Operand 2											
Multiply	Cond				0	0	0	0	0	0	A	S	Rd				Rn				Rs				1	0	0	1	Rm			
Multiply Long	Cond				0	0	0	0	1	U	A	S	RdHi				RdLo				Rn				1	0	0	1	Rm			
Single Data Swap	Cond				0	0	0	1	0	B	0	0	Rn				Rd				0	0	0	0	1	0	0	1	Rm			
Branch and Exchange	Cond				0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	Rn			
Halfword Data Transfer: register offset	Cond				0	0	0	P	U	0	W	L	Rn				Rd				0	0	0	0	1	S	H	1	Rm			
Halfword Data Transfer: immediate offset	Cond				0	0	0	P	U	1	W	L	Rn				Rd				Offset				1	S	H	1	Offset			
Single Data Transfer	Cond				0	1	I	P	U	B	W	L	Rn				Rd				Offset											
Undefined	Cond				0	1	1																					1				
Block Data Transfer	Cond				1	0	0	P	U	S	W	L	Rn				Register List															
Branch	Cond				1	0	1	L	Offset																							
Coprocessor Data Transfer	Cond				1	1	0	P	U	N	W	L	Rn				CRd				CP#				Offset							
Coprocessor Data Operation	Cond				1	1	1	0	CP Opc				CRn				CRd				CP#				CP			0	CRm			
Coprocessor Register Transfer	Cond				1	1	1	0	CP Opc			L	CRn				Rd				CP#				CP			1	CRm			
Software Interrupt	Cond				1	1	1	1	Ignored by processor																							

Thumb State		ARM7 State	
R0	→	R0	Lo Registers
R1	→	R1	
R2	→	R2	
R3	→	R3	
R4	→	R4	
R5	→	R5	
R6	→	R6	
R7	→	R7	
		R8	High Registers
		R9	
		R10	
		R11	
		R12	
SP (R13)	→	SP (R13)	
LR (R14)	→	LR (R14)	
PC(R15)	→	PC(R15)	
CPSR	→	CPSR	
SPSR	→	SPSR	

Fig. 10 Thumb and ARM register maps

processes. Therefore, we will primarily focus on how a RISC-based architecture handles the expansion of the 16-bit architecture to 32 bits, and how the 32-bit architecture was expanded into 64 bits.

The AARCH64 architecture aims for new markets, such as servers, scientific processing, machine learning, and more. So, on top of supporting larger address space (64 bits), it supports many new technologies and can also be extended in the future.

Few of the unique features of the AARCH64 ISA are:

- It uses a new instruction set, A64.
- Uses 31 general-purpose 64-bit registers.
- The program counter (PC) is no longer directly accessible as a general-purpose register.
- Instructions are still 32 bits long and mostly the same as A32, but most conditional execution dropped.
- Can paired loads/stores.
- Most instructions can take 32-bit or 64-bit arguments.
- 64-bit address space.
- Support single- and double-precision FP (Fully IEEE 754 complaint).
- Support for AES encrypt/decrypt and SHA-1/SHA-2 hashing instructions.
- Advanced SIMD (Neon) enhanced (ARM 2021).

On the other hand, a few technologies were dropped, such as

- The support for Thumb.
- Predications.

The AARCH64 uses a handful of instruction formats (see Table 8) that help keep a simple instruction decoding scheme but do not allow it to maintain compatibility with previous generations.

Summary of ARM ISA

ARM is the most commonly used ISA today. The company's business model targets the core IP and not the end product, meaning that it is assumed that other companies will license the IP and will build the system out of it. As a result, ARM ISA, on the one hand, is simple but, on the other hand, allows the ISA to be extended at the system level by adding accelerators and other system-level components.

ARM develops a wide range of processor classes that are not always compatible to support different markets and needs. For example, there are three different ISAs, the Thumb, the AARCH-32, and the AARCH-64. These ISA architectures present quite a few fundamental differences, at all levels such as

- Width of data paths
- The number of registers
- Use of predicates (conditions)
- Extensions for security
- Support for virtualization
- Many more

When looking at the system-level support of different ARM cores, there are significant differences between chips belonging to a different group; for example, processors in the R series support faster support for Interrupts to support real-time applications. In addition, the definition and the implementation of the TrustZone (security extension) are different for Series M and the Series A.

Research comparing the ARM ISA to X86 ISA [Ark (2017)][Akr (2019)] did not find fundamental differences which are related to the ISA between these alternatives in terms of performance. But when comparing the power consumption and the design complexity of these alternatives, the use of fixed-size instruction format used to have a considerable advantage, mainly when attempting to decode several instructions in parallel. This is one of the major reasons ARM is more popular than Intel in edge and mobile devices.

The RISC-V Approach for ISA

RISC-V is an open ISA that attracts many researchers and industries. To get around the scalability issue, RISC-V came out with a unique approach; it defines a basic ISA that resembles MIPS architecture and a set of extensions, each aim to meet

Table 8 AARCH64 instruction formats

Type	Bit 31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved	op0			0000				op1									Immediate constant 16 bits															
Unallocated				0001																												
SVE Instructions				0010																												
Unallocated				0011																												
Data Processing – Immediate PC-rel.	op	immlo		10000				immhi																				Rd				
Data Processing – Immediate Others	sf			100			01-11																					Rd				
Branches + System Instructions	op0			101			op1																					op2				
Load and Store Instructions	op0				1	op1	0	op2			op3										op4											
Data Processing – Register	sf	op0		op1	101			op2									op3															
Data Processing – Floating Point and SIMD	op0				111			op1		op2				op3																		

different market segments. As a result, all RISC-V processors must support the common basic ISA, but can still be optimized to meet specific goals.

RISCV: Basic ISA (RISCV 2021)

Although RISCV is based on RISC architecture, it supports two models (RV32I and RV64I) and supports different instruction lengths, as indicated in Fig. 11. This trade-off allows a relatively simple hardware decoder while maintaining an efficient code size and extensibility.

The "basic RISC-V model" (for both models) supports integer-only operations; it exposes to the user 31 general-purpose registers (x1-x31), a single register, x0, which is hardwired to "0" and a special register, PC (program counter) which is dedicated for pointing at the instruction needed to be decode/execute next. All registers are of a size of XLEN (32 or 64).

Four basic formats are being used, as shown in Fig. 12.

For some instructions, the immediate field is further partitioned to indicate a special functionality. For example, the instructions SLLI (logical left shift) and SRLL (logical right shift) differ in their 30th bit, and SLLI instruction and the WSLLI (logical left shift on a 32-bit register) differ by their Opcodes and bit 25 that if zero, it indicates that the instruction belongs to RV32I.

RISCV: Extensions (RISCV 2021)

The RISCV open specification project presents a clean and simple model for the "basic model" that all RISCV architectures must implement. Thus, it allows for extending the basic model with "extensions." Of course, the community must approve these extensions and be uniform among all processors implementing

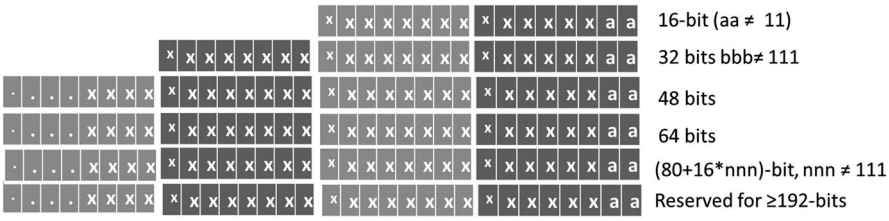


Fig. 11 Instruction decoding lengths

31 25	24 20	19 15	14 12	11 7	6 0	
Funct7	RS2	RS1	Funct3	rd	OPcode	R-Type
Imm[11:0]		RS1	Funct3	Rd	OPcode	I-Type
	RS2	RS1	Funct3	Imm[4..0]	OPcode	S-Type
Imm[3:12]				Rd	OPcode	U-Type

Fig. 12 RISCV integer subsystem format

them. But a RISCV developer can decide what extensions to implement and what extensions are not needed for the specific product or market the core is aiming at. The advantage of this approach is that it allows the core architect to assemble a core that best fits its needs in terms of power, area, and performance concerning the product's specific needs. It also allows maintaining compatibility with respect to a specific extension. The downside of this approach is that it creates software incompatibility between different cores. Hence, core and code you develop for one market most likely could not be used as part of another market.

For the reader's benefit, we chose a handful of examples of extensions out of a larger list of options discussed in the RISCV committee.

Extension M: Integer Multiplication and Division
This extension allows extending the basic integer operations with multiplication and division operations. It uses I-type format.

- MUL performs an XLEN-bit×XLEN-bit multiplication. The spec also defines MULH, MULHU, and MULHSU.
- MULW (RV64 only) multiplies the lower 32 bits of the source registers, placing the sign extension of the lower 32 bits of the result into the destination register.
- DIV and DIVU perform signed and unsigned integer division of XLEN bits by XLEN bits. The instruction is defined for RVI32 and RVI64 bits.

Extension A: Atomic Instructions
RISCV maintains a relaxed memory model, so a FENCE instruction needs to be used to impose ordering and overall consistency among events. In case of atomic operations are needed, the spec defines the A-extension.

At the ISA level, the atomicity is defined by using predefined two bits, aq (acquire access) and rl (release access). A more comprehensive description of these operations can be found in [Sar (1990)][Gha (1990)].

The A extension also defines the Load-Reserved (LR) and Store-Conditional (SC) Instructions and a set of atomic memory operations (AMO) that performs read-modify-write operations:

- AMOSWAP.W/D
- AMOADD.W/D
- AMOAND.W/D
- AMOOR.W/D
- AMOXOR.W/D
- AMOMAX[U].W/D
- AMOMIN[U].W/D

Extension F: Single-Precision Floating Point
The RISCV extension spec allows adding a subsystem to deal with floating-point arithmetic.

Unlike previous generations of RISC processors (mainly for 32 bits) that treat floating point as an accelerator, the RICV allows including FP operations as part of

the ISA and handling them as a separate execution pipe which can be implemented using a separate hardware module.

The F extension assumes a dedicated floating-point registers file, containing 32 registers f_0–f_{31}, each FLEN bits wide, and a floating-point control and status register ***fcsr***. FLEN is defined to be 32 in the case of single-precision FP, 64 in the case of double-precision FP and 128 in the case of quad-precision FP.

The extension defines special instructions to load and store data from/to memory to FP register file. The spec also defines basic operations such as FADD.S, FSUB.S, FMUL.S, and FDIV.S operations to perform single-precision floating-point addition, subtraction, multiplication, and division between rs1 and rs2, writing the result to rd. It also defines more sophisticated instructions such as FMIN.S and FMAX.S or the FSQRT.S, computes the square root of rs1, and writes the result to rd.

Summary of ISA Selection

The ISA is the layer that connects the software layer and the hardware implementation. In this chapter, we described and discussed three approaches for selecting ISA. The CISC tries to provide the maximum expressive power to each instruction, the RISC approach calls to simplify the ISA to allow more software optimizations (mainly due to the use of three-operands instructions and to accelerate the speed of the processor). Finally, the RISCV approach calls for a standard "basic ISA" architecture with extensions that aim to solve specific needs.

Each of these approaches has its advantages and disadvantages. The use of CISC architecture eases the process of maintaining backward compatibility, with the cost of efficiency of the decoding process and the increasing cost of doing parallel decoding. On the other hand, using RISC can improve performance and enable simple parallel decoding. But, this approach makes backward compatibility to become very difficult. Using the RISCV approach seems to be a good trade-off between the RISC and the CISC approach since it can maintain compatibility (at least for the basic ISA) and achieve good scalability and performance. Still, it causes an inherent SW compatibility challenge between cores that use different extensions since the software that runs on one processor may not be able to run on other processors.

Vector and SIMD Extensions

A single dimension array of elements, AKA Vector, and an n-dimensional array of elements (AKA n-dimension Matrix) are fundamental data structures in almost all scientific applications. For example, research shows 80–90% of the time, convolutional neural networks (CNN) execute matrix-related operations. Most operations on these data structures can be done in parallel. The best way to expose it to the hardware is by using SIMD (Single Instruction Multiple Data) or vector operations,

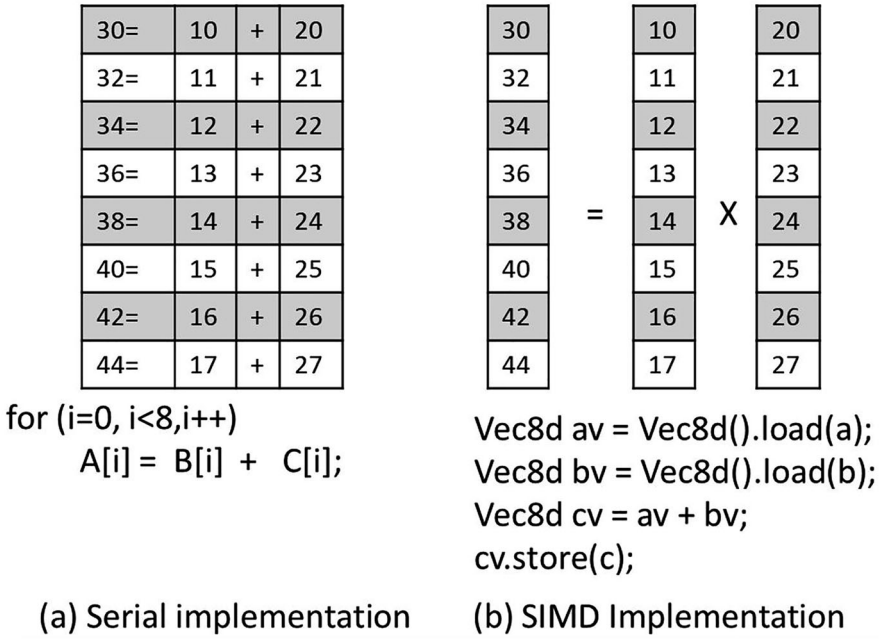


Fig. 13 Serial vs. SIMD operation

as indicated in Fig. 13. SIMD saves extra decoding power, complexity, and the need to verify data dependencies. The SIMD usually works on dedicated vector registers and has a fixed number of elements it affects. As a result, the instruction code for adding four elements differs from the instruction code that works on eight elements. Vector extensions, on the other hand, usually operate on memory and can have a variable size. This subsection will discuss these two forms of operations on n-dimensional arrays of elements.

SIMD Architectures

SIMD (Single Instruction Multiple Data) presents a simple way to expose parallel execution to the hardware. The SIMD assumes that the SW guarantees that no dependencies exist between the operation. In return, the architecture ensures that all operations be executed in parallel, as indicated by Fig. 14, which shows an addition operation using arrays of four elements.

Although many architectures adopt SIMD operations, this chapter focuses on Intel's family of SIMD instructions. In order to support that, Intel architecture extends the ISA to include vector registers. The different generations of SIMD operations use a different number of elements in each Register, as indicated in Fig. 15.

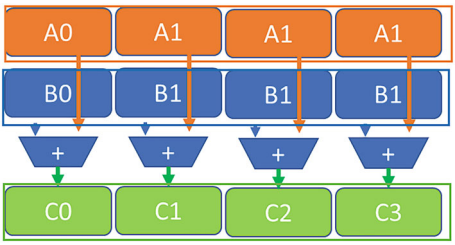


Fig. 14 Four-way SIMD ADD operation

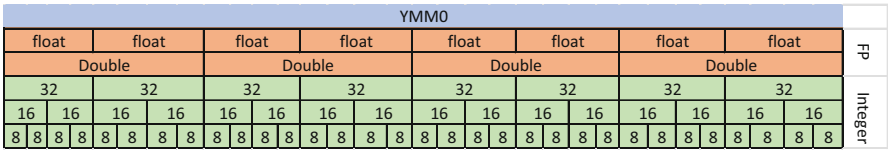


Fig. 15 SIMD register structures

MMX

MMX ISA extension was the first SIMD instruction set that Intel added to their core (Pentium P5) to support multimedia applications (Peleg and Weiser 1996). MMX defines eight new registers, named MM0 through MM7, 64 bits wide that can store two 32-bit integers, four 16-bit integers, or eight 8-bit integers in parallel (see Fig. 15). MMX was mainly targeted to compete with multimedia acceleration cards, such as audio cards that were very popular that days. Thus, MMX supports integer operations only, but the user could use fixed-point math if floating-point operations were needed. Such support is sufficient for general digital signal processing applications and audio-related operations.

To support fast context switch and fast response time to interrupt the MMX registers are aliases for the existing x87 floating-point unit (FPU) registers. At the event of a context switch, the entire state of the task needs to be saved. By sharing the FP and MMX states, Pentium cores support efficient context switch and interrupt service operations.

Streaming SIMD Extensions (SSE)

SSE extends the MMX operations with about 70 new instructions, most of which work on single-precision floating-point data. Unlike the MMX that reuses existing x87 floating-point registers, SSE is based on a separate register file, allowing that FP unit to work parallel to the SIMD unit. The SSE has a few versions, for example, SSE2, SSE3, and SSE4; each generation extends its preceding versions. The use of floating points allows the SSE to extend the support to a broader range of applications.

SSE initially added eight new 128-bit registers known as XMM0 through XMM7. The x86-64 bits instruction set added a further eight registers XMM8 through XMM15.

SSE used only a single data type for XMM registers:

- Four 32-bit single-precision floating-point numbers SSE2 expands the usage of the XMM registers to include either of the following:
  - Two 64-bit double-precision floating-point numbers
  - Two 64-bit integers
  - Four 32-bit integers
  - Eight 16-bit short integers
  - Sixteen 8-bit bytes or characters

Advanced Vector Extensions (AVX)

AVX (Intel 2021a, b) uses 16 YMM registers (see Fig. 15) to perform SIMD operations on eight 32-bit single-precision floating point numbers or four 64-bit double-precision floating point numbers. Although registers were extended from 128 to 256 bits long, legacy SSE instructions are supported and are being operated on the lower 128 bits of the YMM registers.

AVX also extends the coding scheme by introducing a three-operand SIMD instruction format called VEX. Under this new scheme, the destination register is distinct from the two source operands. For example, an SSE instruction using the conventional two-operand form a = a + b is now being extended to three-operand to form c = a + b.

AVX was extended by the AVX2 standard introducing the notion of FMA (Fused Multiplication Add), and also providing special attention to an efficient data movement between memory and registers and between registers.

For example

Instruction	Description
VBROADCASTSS, VBROADCASTSD, VBROADCASTF128	Copy a 32-bit, 64-bit, or 128-bit memory operand to all elements of a XMM or YMM vector register.
VMASKMOVPS, VMASKMOVPD	Conditionally reads elements from a SIMD vector memory operand into a destination register
VPERM2F128	Shuffle the four 128-bit vector elements of two 256-bit source operands into a 256-bit destination operand
VPSRAVD	Shift right arithmetically. Allows variable shifts where each element is shifted according to the packed input.

AVX-512 extension is the latest version of the advanced ISA for SIMD operations Intel has in that family. AVX-512 extends the registers to 512 bits long, adding hundreds of new instructions supporting many different options for using wide vector operations. The standard of AVX-512 suggests multiple possible extensions

to the AVX. It defines a subset AVX-512F (AVX-512 Foundation) and several suggested extensions. A few of these extensions are:

- AVX-512 Foundation (F) – presents the EVEX coding scheme to support 512-bit registers and operations. It also defines options for using masks, broadcast parameters, how to make the rounding, what exceptions can occur, and more.
- AVX-512 Conflict Detection Instructions (CD) – provides hardware support for conflict detection. It is mainly gusseted to be used to allow more loop optimizations and to support transaction-memory like operations.
- AVX-512 Prefetch Instructions (PF).

The AVX-512 was so complicated that Intel decided to drop much of it from its current generation of CPU, Alder Lake (Ian Cutress 2021), but we may expect it to come back in the future.

Support for Machine Learning

The trend of using accelerators to support new applications and markets continues, and so, recently, Intel added new technologies to support machine learning operations. The current ISA extensions Intel suggests to support such an environment include the AVX-VNNI and the AMX (Intel, May, 2021b).

The VNNI extends the core with vector operations, while the AMX provides direct support for matrix operations that many machine learning applications need.

These technologies are integrated as part of the general-purpose out-of-order pipeline of the processors, maintaining memory coherency and supporting other Intel technologies (see Fig. 16).

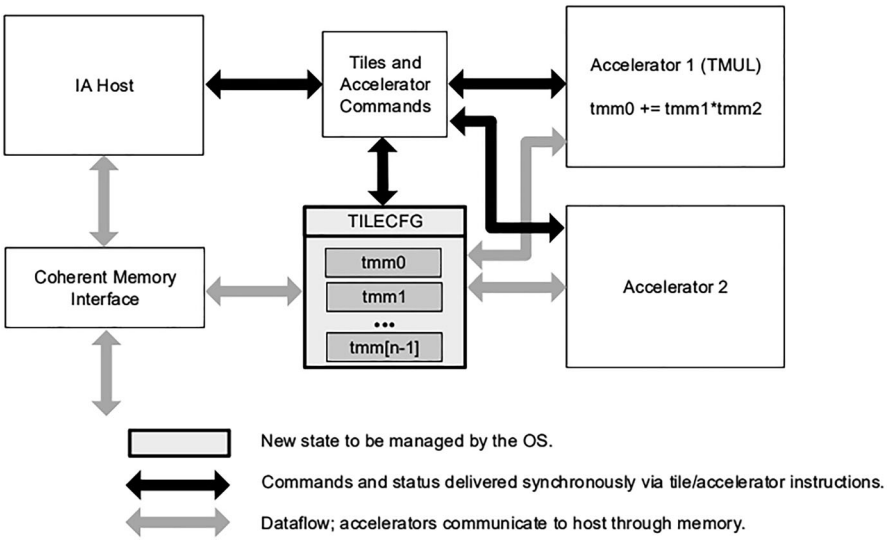


Fig. 16 AMX system architecture

The XSAVE instruction supports the saving and the restoring operations of the AMX internal state.

Discussion on the Use of SIMD Operations (in Intel's Cores)

Using SIMD operations can significantly improve performance with a relatively small penalty in terms of power. Thus, the use of SIMD instructions is quite common and is used by all architectures (Zhang et al. 2021; ARM 2021; IBM 2021).

Although SIMD shows excellent potential, it has its limitations, such as:

- It consumes a relatively large area but benefits only applications that can take advantage of it; as Amdahl low indicates (Amdahl 2013), the overall performance of new technology is limited to the frequency that feature is being used. Thus, it may be very beneficial for some applications, but a loss of opportunity for applications that do not use it. Thus, it is not always clear if adding such a new feature, mainly for general-purpose computers, is worth the overhead.
- SIMD instructions have a fixed size. Thus, you need to recompile your code depending on the hardware you use.
- Managing the vector register file is quite complicated.

The SIMD approach Intel took also presents the forward compatibility issue. A programmer needs to rewrite her application each time that technology changes. As will be indicated when we discuss the solutions taken by CUDA's developers (see section "HW/SW Codesign: The CUDA Approach"), a different approach can ease these issues.

Support for Vectors

Direct support for vector operations started appearing in the early 1970s; as appear in the TI-ASC (Watson 1972). But CRAY-1 (Russell 1978) is the first commercial system that can also be considered a commercial success. CRAY-1 was designed as a fast scalar machine, and the vector support was added to it as an add-on unit. Thus it could run vector and nonvector applications efficiently.

Espasa et al. (1998) provides a comprehensive comparison of all the different vector machines in the market between 1972 and 1996 (see Table 9).

Vector architectures are similar to SIMD operations since both support mathematical operations on n-dimensional arrays of elements, but there are a few important differences between these two technologies, such as:

- SIMD performs operations between registers, while vectors can perform the operations directly on memory (at least from the user perspective). Please note that at the micro-operation level, vector operations are often divided into three pipelined stages:
  - Bring data from memory to very long registers. The data can be located in noncontinuous locations in the memory. In this case, a gather operation may be needed.

Table 9 Vector machines 1972–1996

Machine	Year intro	Vector registers	Elements/ Register	pipes	Flops/cycle	Words/cycle
TI-ASC	1972	–	–	4	4	4 (32th)
STAR-100	1973	–	–	1	2	3
CRAY-1	1976	8	64	1	2	1
Fbjitsu VP200	1982	256-8	1024-32	2	4	4
Cray X-MP	1983	8	64	1	2	2 + 1
Hitachi SSIO/ZC	1983	32	256	2	12	8 or 2
NEC SX-2	1984	8 + 8K	256/64-256	4	16	8 or 4
CRAY-2	1985	8	64	1	2	1
Hitachi S820/80	1987	32	512	4	12	8 or 4
Cray Y-MP	1988	8	64	1	2	2 + 1
Fujitsu VP2600	1989	2048-64	64-2048	4	16	8
NEC SX-3	1990	8 + 16K	256/64-256	4	16	8 + 4
Cray C90	1992	8	128	2	4	4 + 2
NEC SX-4	1996	8 + 16K	256/64-256	8	16	16

  - Perform the math operations (most of the time SIMD operations are being used for that).
  - Write the data back to memory (scatter may be needed for that).
- From the ISA perspective, the same instruction format is used to handle vectors of different sizes. It helps to maintain backward compatibility and the use of libraries that will be agnostic to the length of the vectors.

Please note that most vector machines use large register files to support vectors that can be managed as compiler-controlled buffers. It allows the system to hide memory latency and leverage memory bandwidth.

Cross-Layers Optimizations

The use of layers of abstractions helps to simplify the development of new systems but has its cost in terms of performance and power consumption. This chapter focuses on computer systems that improve system performance and power by making cross-layers optimizations, meaning breaking the layered structure to improve performance and/or power. We start this chapter with a short introduction that provides a brief background on how "traditional" systems are implemented and how the architecture-abstraction and microarchitecture abstraction communicate. Then, we extend the discussion to architectures that change these interfaces, for example, allowing microcode programming, VLIW (very long instruction word), exposed-pipeline DSP architectures, using dataflow-based architectures, and more.

Background

When designing a CPU, we usually distinguish between the memory/logic elements, the control path, and the data path (see next chapter). The memory elements keep data and instructions; the logic elements perform the required operations; the data path allows data and instructions to move from one element to another and from one execution stage to another; and the control response determines what will be performed and when.

Simple instructions, such as arithmetic operations between registers, can be implemented using a relatively simple control structure. But when more complicated instructions are involved, and multiple instructions are allowed to be fetched, decoded, and executed in parallel, the control structure can be pretty challenging.

Delayed Branch in MIPS

MIPS architecture uses delayed-branch slots, a simple way to expose microarchitecture structure to the compiler (Hennessy et al. 1982; McFarling and Hennesey 1986). In a pipelined implementation of MIPS architecture, the system can verify if the branch was taken or not only X cycles after the instruction was fetched. During that time, the system can either stall until the conditional branch is resolved or speculatively fetch the instruction and recover if the assumption was found to be wrong. The branch-slot method suggests filling up these speculative slots with instructions that are not dependent on the outcome of the branch (See Fig. 17).

To take advantage of the delay-slot feature, the architecture needs to expose to the compiler how many slots are available (this parameter is machine-dependent), and the compiler needs to re-order the instructions to use it. In the case that the compiler cannot find enough "independent" instructions, it needs to insert NOOP (no-operation) instructions to the designated slots.

Although the use of branch slots was proven to be very effective, it suffers from different downsides, such as (a) code needs to be recompiled for architectures with varying numbers of branch slots, (b) it does not work well with some advanced technologies, such as out-of-order architectures. As a result, advanced versions of MIPS stopped supporting delayed-branch slots.

```
R1 = 10
R2 = 20
R5 = 30
If (R3>0)
   Do something1
else
   do something else
```

(a) W/O delay branch

```
R1 = 10
If (R3>0)
      R2 = 20  \\Delay slot #1
      R5 = 30  \\Delay slot #1
   Do something1
else
   do something else
```

(b) with 2 delay branches

Fig. 17 Branch delay slot

The User-Defined Microcode Programming

Micro-operations are a common way to manage the resources of a system. Maurice Wilkes (Wilkes 1951) suggested using microcode to control the operation of a "smart calculator", but basic methods have been used since. Current systems, such as Intel Alder-Lake (Rotem et al. 2022), still use micro-operations and even built a special cache to keep CISC instructions in their decoded format (Solomon et al. 2001). Due to security reasons, most of the processors allow changing the microcode of the system only by the manufacturer. But few distinct architectures allowed applications and even user-level code to modify the microcode. For example, IBM allowed applications running on the system-360 to control the microcode (Tucker 1967), Burroughs Corporation ((Burroughs Corporation 1972), (DeWitt et al. 1973)) allowed users to adopt their system behavior to the specific applications they are running. Digital Corporation (Badeau et al. 1992) came up with a different approach that allowed a user to extend the ISA via a special reserved area in the microcode ROM.

This trend is not widely used anymore since it causes compatibility issues, requires special support from the compiler, and often exposes the hardware to faults and security hazards.

VLIW Architectures

VLIW is another method to allow user-level applications to control the behavior of the microarchitecture. Here, the system does not intend to enable changing the ISA or creating new instructions. However, it aims to allow the compiler to determine what operations the system could execute in parallel and what operations still need to be executed sequentially due to data or control dependencies (see Fig. 18)

The VLIW is one of the first systems that was developed as a SW/HW codesign. The primary enabler of the architecture was advanced compiler techniques (Rau and Fisher 1993), such as the use of "percolation" (Foster and Riseman 1972) and trace schedule (Fisher 1981) that enabled the creation of the VLIW architectures, such as the ELI-512 (Fisher 1983).

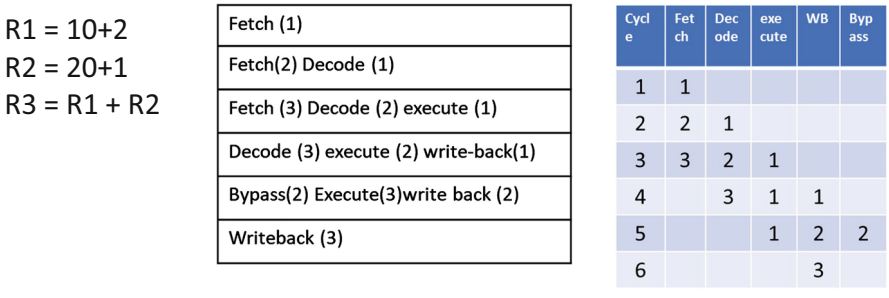


Fig. 18 Structure of a VLIW architecture

Although the architecture was considered very promising, it almost disappeared a few years later. The main reason was the need to recompile the code each time the architecture changed. Although recompiling a code may sound like an easy task, it requires a deep understanding of the application details, architecture details, and a lengthy testing process that can be costly.

Intel Itanium™ architecture (Sharangpani 1999a, b) was an attempt to create a VLIW architecture that could scale and could maintain backward compatibility. Thus, since it fails to become a commercial success, we will not extend the discussion on this architecture.

Today, VLIW is still widely used in mainly two market segments; DSP architectures (Karthihaa et al. 2021) and accelerators (Traore et al. 2022). DSP architectures were initially aimed at the signal-processing market. For this market, efficient power consumption is critical, and the code created can be characterized as having vast internal parallelism that the compiler can take advantage of. More than that, traditionally, the code is tailored for each architecture to extract maximum performance at the lowest power; advanced techniques such as using exposed pipe are commonly used. As a result, recompile the code and test it to verify that it meets the real-time and power consumption constraints traditionally done for each application and system, so compatibility is less of an issue for these systems.

Accelerators, such as graphics cards (Mantor 2012) of machine learning accelerators (Intel 2017), are tailored to execute code most efficiently. Most such systems use intermediate representation code (IR); for example, PTX for Nvidia architectures, so the optimization for a specific architecture is done at run time, using Just-In-Time (JIT) technology. Thus, the core can be implemented using VLIW technology in order to improve power and performance.

HW/SW Codesign: The CUDA Approach

CUDA – Compute Unified Device Architecture (NVIDIA 2007; Hwu et al. 2022) presented a new approach to how programs and hardware should be developed. The new approach calls for HW/SW codesign that considers the characterization of the domain it targets and adjusts it to the characterization of the GPU Nvidia built. As before, this chapter does not intend to cover the entire history of CUDA to provide a comprehensive list of the features CUDA supports. Still, it aims to focus on its main contribution to redefining the interfaces between the application, the architecture, and the microarchitecture.

To achieve this goal, Nvidia focused on applications with vast parallelism. As a result, CUDA supports only a limited number of data structures: scalars, 1D, 2D, and 3D arrays of memory. A CUDA code describes the operations that the processor executes as well as the Code that the Accelerator (GPU) runs (see Fig. 19).

In the past, the user had to move data between the CPU's main memory to the GPU's main memory. Today, due to the new virtual shared memory technology, the system will manage the location of the data automatically and move it if needed.

Code compile & transferred To GPU

```
__global__ void cuda_hello1(){ printf("Hello World from GPU1!\n");
}
```

Code compile & transferred To GPU

```
__global__ void cuda_hello2(){ printf("Hello World from GPU2!\n");
}
```

Host Code

Call GPU Code

Host Code

Call GPU Code

Host Code

```
int main()
{ printf("Hello from CPU1 \n");
  cuda_hello1<<<1,1>>>();
  printf("Hello from CPU2 \n");
  cuda_hello2<<<1,1>>>();
return 0; }
```

Fig. 19 CUDA code

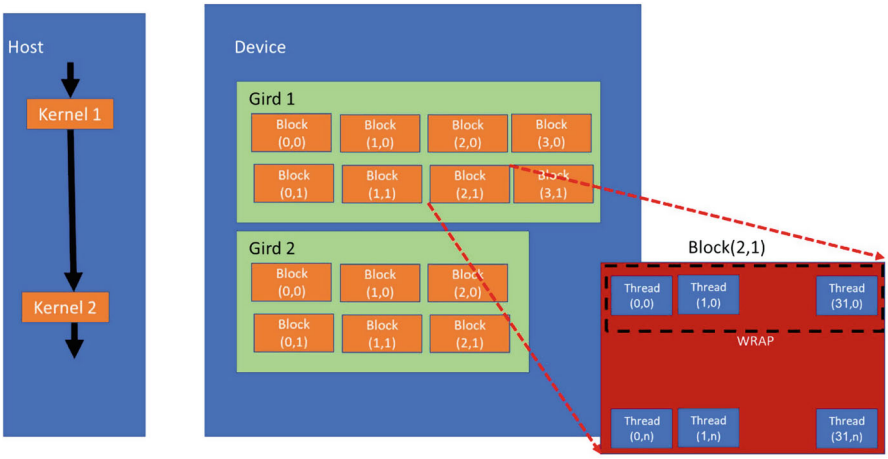


Fig. 20 Memory organization in CUDA

Thread in CUDA differs from the notion of threads in general-purpose computers. In CUDA, a thread usually represents the operation that the GPU needs to perform on a single data point. Thus, the same code can be developed for any size of an array. The overall structure of the data used during the execution of a kernel in the GPU is termed Grid, and is divided into blocks (see Fig. 20). A block is assigned to an SM (a processor) and will not migrate from it. Thus, no coherency is needed during the execution time of this kernel. A thread manipulates each data

element within a block. All threads handed data points belonging to the same block can share information, but threads belonging to different data blocks cannot share data. This SW/HW codesign can be used to simplify the hardware and for using other mechanisms that otherwise could not be considered.

CUDA presents the notion of two parallelism levels; the grid's partition into independent blocks, as depicted in Fig. 20 serves as the higher level of parallelism. However, the system also has a hardware mechanism capable of collecting all independent instructions located at the same IP within a block (but in different threads) into a special WRAP structure and executing it in an execution unit resembling SIMD. This capability is the lower level of parallelism, and Nvidia calls this execution a SIMT mode. NVIDIA distinguishes between SIMD and SIMT modes; under SIMD the same operation is being broadcast by different execution units, and the operation is done on different data items. In SIMT (single instruction multiple threads), multiple threads are being executed in a lock-step manner, but a predicator controls each thread, so not all threads may execute the same instruction all the time.

Nvidia uses CUDA to target different markets and to support different generations of GPU cards, each of them may target different markets and may need different characterization of the hardware and the software. To cope with that (1) Nvidia added the notion of CUDA capabilities to indicate what features are supported (WIKIPEDIA 2022), (2) CUDA uses PTX as an intermediate representation (IR) which is translated to the features and the assembly of the specific device, and (3) the different Nvidia graphics card does not guarantee to be backward-compatible in terms of power and performance. An application that runs on a newer version of GPUs is not guaranteed to preserve or improve performance with respect to running the same code on older architecture.

ISA Agnostic Systems

So far, we have mainly focused on the relations between the ISA and the implementation as represented by the microarchitecture. In this chapter, we will look at solutions that aim to overcome the limitations an ISA presents.

The Use of Intermediate Representations

Java was one of the first systems to use a virtual machine (Lindholm and Yellin 1996) that uses an intermediate representation, called bytecode, as the ISA to be compiled and optimized (Albert et al. 2007). Sun Microsystems invented Java and described the language as "write once run anywhere"; meaning that the same intermediate representation (IR) of a code could run on any hardware.

To achieve that, Java source code is compiled to a virtual machine, and each architecture needs to compile or Just-In-Time (JIT) translation the code from the

virtual machine IR into the actual assembly code that can run on the specific hardware. Since architectures are very different regarding the number of registers, ISA, etc., Java virtual machine decided not to target any existing architectures. Instead, it decides not to use any register but to adopt the Polish notation, meaning all operations are transferred using the struck memory (Bredlau and Deremer 2001). Java also decided not to allow direct system calls or any other resources of the actual machine. Instead, it provides an interface to communicate with the actual operating systems and resources.

Today the notion of using a virtual machine and intermediate representation is quite common for many systems, although many times for different reasons; a few examples are as follows:

- LLVM (Sarda and Pandey 2015), one of the most commonly used compilers, uses IR to represent the results of the compilation of many different programming languages so that it can run modules written in different programming languages.
- Nvidia uses the PTX IR (NVIDIA 2022) to unify all the different GPUs they need to support.
- OMNX (OMNX 2020) is an open specification consisting of three main components: (1) an extensible computation graph model, (2) standard data types, and (3) built-in operators. OMNX intends to unite the definition of other IR of different environments that aim to support neural network applications.
- Python (VanRossum and Drake 2010) and other scripting languages are using virtual environments and IR.

Binary Translation

In 2005, Apple decided to change the main CPU on their laptops and desktop from IBM to Intel. This strategic move was enabled by the use of a binary translation SW layer called Rosetta. Recently, Apple decided to make another transition and to move from Intel-based processors to ARM-based cores. This transition was also supported by a newer version of the binary translation code, Rosetta-II (Apple 2021; Dalakoti and Chakraborty 2022).

Apple was not the first company that use binary translation, but most other companies used it to run multi-ISA Software on their architecture. To list a few of them:

- Digital-corporation started using binary translation systems during the 70th and early 80th. The FX! (Hookway and Herdeg 1997) aimed to run X86 code on VAX machine, and the AXP aimed at running VAX and MIPS Code on the newly Alpha processor. Although the commercial success of these systems was limited, it enabled the development of other binary translation systems that showed to be more mature and could achieve better performance.

- Transmeta was a company that aimed to run X86 code on top of VLIW, RISC-based architecture, to create a low-power X86 core (Klaiber 2000). The company produced two cores named Crusoe and Efficeon but had only limited success.
- The System As a Service (SAS) that cloud computers provide is based on the ability of the host architecture to run any code that was compiled for any other ISA (Smith and Nair 2005).

Summary

This chapter described different approaches for developing new architecture and new ISA. The chapter shows that commercial reasons, such as compatibility, usage models, low power core, and more, determine what ISA best fits the market's needs and will hopefully become commercially successful.

As this chapter indicated, it is extremely important to distinguish between general-purpose computers and dedicated, domain-specific architectures. General-purpose architectures need to take care of scalability, backward compatibility, different types of application optimizations, etc. But, domain-specific architectures can rely on the characterization of the application or the market they are currently targeting.

As the last section indicates, the importance of the ISA and the backward compatibility reduces over time since modern processors have enough computing power to spar to minimize the impact of overhead created by the binary translation phase.

But, when looking at domain-specific systems and systems with special needs such as low power or the use of massive networking and vast physical memory, the ability to adjust the ISA and the potential extensions of the architecture are still playing a significant role in achieving the goals of the system.

References

Albert E, Arenas P, Genaim S, Puebla G, Zanardini D (2007) Cost analysis of Java bytecode. Programming Languages and Systems, pp 157–172

AMD (2000) The AMD x86-64 architecture. Retrieved from http://www.x86-64.org/

Amdahl GM (1967) Validity of the single processor approach to achieving large scale computing capabilities. In: Spring Joint Computer Conference, pp 18–20

Amdahl GM (2013) Computer architecture and amdahl's law. Computer:38–46

Apple (2021) Rosetta 2 binary translation comprehensive supported instruction set list. Retrieved from https://developer.apple.com/forums/thread/653902

ARM (1995) ARM7TDMI data sheet. Retrieved from https://www.dwedit.org/files/ARM7TDMI.pdf

ARM (2021) Introducing NEON development. Retrieved from https://developer.arm.com/documentation/dht0002/a/Introducing-NEON/What-is-NEON-

Badeau RW, Bahar R, Bernstein D, Biro L, Bowhill W, Brown J et al (1992) A 100-Mhz micropipelined VAS microprocessor. IEEE J Solid-State Circuits:1585–1598

Bredlau C, Deremer D (2001) Assembly language through the Java virtual machine. In: Proceedings of the Thirty-Second SIGCSE Technical Symposium on Computer Science Education, pp 194–198
Burroughs Corporation (1972) Burroughs B-1700 software operational guide
Dalakoti V, Chakraborty D (2022) PPLE M1 chip vs intel (X86). EPRA Int J Res Dev:207–211
Dennard RH, Gaensslen FH, Yu H-N, Rideout VL, Bassous E, LeBlanc AR (1974) Design of ion-implanted MOSFET's with very small physical dimensions. IEEE J Solid State Circuits:256–268
DeWitt DJ, Schlansker MS, Atkins DE (1973) A microprogramming language for the Burroughs B1726. In: Workshop of microprogramming, pp 21–29
Engineering S (1991) Makimoto's wave. Retrieved from https://semiengineering.com/knowledge_centers/standards-laws/laws/makimotos-wave/
Espasa R, Valero M, Smith JE (1998) Vector architectures: past, present, and future. In: ICS. ACM, pp 13–17
Fisher JA (1981) Trace scheduling: a technique for global microcode compaction. IEEE Trans Comput:478–490
Fisher JA (1983) Very long instruction word architectures and the ELI-512. In: 10th Annual International Symposium on Computer Architecture, pp 140–150
Flynn MJ (1972) Some computer organizations and their effectiveness. IEEE Trans Comput:948–960
Foster CC, Riseman EM (1972) Percolation of code to enhance parallel dispatching and execution. IEEE Trans Comput 21(12):1411–1415
Furber SB (2000) ARM system-on-chip architecture. Pearson Education
Garner RB (1988) The scalable processor architecture (SPARC). In: COMPCON Spring 88 Thirty-Third IEEE Computer Society International Conference. IEEE, pp 3–31
Gustafson JL (1988) Reevaluating Amdahl's law. Commun ACM:532–533
Hennessy JL, Patterson DA (2007) Computer architecture: A quantitative approach. Morgan
Hennessy J, Jouppi N, Przybylski S, Rowen C, Gross T, Baskett F, Gill J (1982) MIPS: a microprocessor architecture. ACM SIGMICRO Newsl:17–22
Hill MD, Marty MR (2008) Amdahl's law in the multicore era. Computers:33–38
Hookway RJ, Herdeg MA (1997) Digital FX! 32: combining emulation and binary translation. Digit Tech J:3–12
Hwu W-MW, Kirk DB, Hajj IE (2022) Programming massively parallel processors – a hands-on approach. Elsevier
Ian Cutress AF (2021) Instruction sets: Alder Lake dumps AVX-512 in a BIG way. Anandtech. Retrieved from https://www.anandtech.com/show/16881/a-deep-dive-into-intels-alder-lake-microarchitectures/5
IBM (2021) Using the SIMD libraries. IBM. Retrieved from https://www.ibm.com/docs/en/xl-fortran-linux/15.1.6?topic=libraries-using-simd
Intel (2017) Intel® MovidiusTM MyriadTM. Retrieved from https://www.movidius.com/myriad2
Intel (2021a) Intel® 64 and IA-32 architectures – software developer's manual. Retrieved from https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf
Intel (2021b, May) Intel architecture instruction set extensions and future features programming reference
Kane G, Heinrich J (1992) MIPS RISC architectures. Prentice-Hall
Karthihaa A, Karthika S, Priyadharshini KM, Sivasankari L, Anand IV, Samuel TA (2021) Design and implementation of VLIW DSP processors for high ended embedded based systems. In: AIP Conference Proceedings
Klaiber A (2000) The technology behind Crusoe processors: low-power x86-compatible processors implemented with Code Morphing software. Transmeta Corp
Lindholm T, Yellin F (1996) The Java virtual machine specification
Macro (1996) Instruction set reference manual. Digital Equipment Corporation

Mantor M (2012) AMD Radeon™ HD 7970 with graphics core next (GCN) architecture. In: IEEE Hot Chips 24 Symposium (HCS), pp 1–35
McFarling S, Hennesey J (1986) Reducing the cost of branches. CM SIGARCH Computer Architecture News:396–403
Neumann J (1945) First draft of a report on the EDVAC. U.S. Army Ordnance Dept. Univ. Pennsylvania Moore, School Elect. Eng.
NVIDIA (2007) NVIDIA CUDA compute device architecture – programming guide. Retrieved from https://developer.download.nvidia.com/compute/cuda/1.0/NVIDIA_CUDA_-Programming_Guide_1.0.pdf
NVIDIA (2022) PTX ISA. Nvidia. Retrieved from https://docs.nvidia.com/cuda/parallel-thread-execution/index.html
OMNX (2020) Open Neural Network Exchange Intermediate Representation (ONNX IR) specification. Retrieved from https://github.com/onnx/onnx/blob/main/docs/IR.md
Patt Y, Patel S (2003) Introduction to computing systems. McGraw-Hill
Patterson DA, Fehr ES, Séquin CH (1979) Design considerations for the VLSI processor of X-TREE. In: Proceedings of the 6th Annual Symposium on Computer Architecture, pp 90–101
Peleg A, Weiser U (1996) MMX technology extension to Intel architecture. Micro 16(4):42–50
Pyeatt LD, Ughetta W (2019) ARM 64-bit assembly language. Newnes
Radin G (1983) The 801 minicomputer. IBM J Res Dev 27(3):237–246
Rau BR, Fisher JA (1993) Instruction-level parallel processing: history, overview, and perspective. J Supercomput 7(1):9–50
RISCV (2021) The RISC-V instruction set manual. Retrieved from https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf
Ronen R, Eliahu A, Leitersdorf O, Peled N, Korgaonkar K, Chattopadhyay A et al (2021) The Bitlet model: A parameterized analytical model to compare PIM and CPU systems. ACM J Emerg Technol Comput Syst:1–29
Rotem E, Yoaz A, Rappoport L, Robinson SJ, Mandelblat JY, Gihon AE (2022) Intel Alder Lake CPU architecture. MICRO:13–19
Russell RM (1978) The CRAY-1 computer system. Commun ACM:63–72
Sarda S, Pandey M (2015) LLVM essentials. Paket
Sharangpani H (1999a) Intel® Itanium™ processor microarchitecture overview. Microprocessor Forum 10(4)
Sharangpani H (1999b) Itanium™ processor microarchitecture overview. Microprocessor Forum
Singh A (1988) The 8088 microprocessor: programming, interfacing, software, hardware, and applications. Prentice-Hall
Smith JE, Nair R (2005) Virtual machine architectures, implementations and applications. Morgan Kaufmann Publishers
Solomon B, Mendelson A, Orenstien D, Almog Y, Ronen R (2001) Micro-operation cache: a power aware frontend for variable instruction length ISA. In: International Symposium on Low Power Electronics, pp 4–9
Traore M, Langlois JM, David JP (2022) ASIP accelerator for LUT-based neural networks inference. In: IEEE Interregional NEWCAS Conference (NEWCAS), pp 524–528
Tucker SG (1967) Microprogram control for system/360. IBM Syst J 6(4):222–241
Turing AM (1938) On computable numbers, with an application to the Entscheidungsproblem. A correction. In: London mathematical society. Oxford Academic, London, pp 544–546
Valiant LG (1990) A bridging model for parallel computation. Commun ACM:103–111
VanRossum G, Drake FL (2010) The python language reference. Python Software Foundation, Amsterdam
Watson W (1972) The TI ASC: a highly modular and flexible super computer architecture. In: Fall Joint Computer Conference, pp 221–228
WIKIPEDIA (2022) CUDA. Retrieved from https://en.wikipedia.org/wiki/CUDA
Wilkes M (1951) The best way to design an automatic calculating machine. In: Computer Inaugural Conference, Manchester.
Zhang Y, Yang W, Li K, Tang D, Li K (2021) Performance analysis and optimization for SpMV based on aligned storage formats on an ARM processor. J Parallel Distrib Comput:126–137

Architectures for Self-Powered Edge Intelligence

3

Amit Ranjan Trivedi, Jaeha Kung, and Jong Hwan Ko

Contents

Evolution of Edge Intelligence and a Pathway to Self-Powered Intelligent Computations . . . 90
Architectures for Energy Harvesting in IoT Edges . . . 92
A Self-Powered Image Sensor System with Autonomous Mode Management (AMM) . . . 93
Factors Affecting Self-Power Performance . . . 94
ROI-Aware Image Processing Architecture . . . 95
Moving Object Detection Architecture . . . 96
ROI-Based Coding Architecture . . . 97
Resource-Aware Control of Target Data Rate . . . 98
Resource-Aware Control of Encoding Data Rate . . . 100
Architectural Support for Handling Sparsity in IoT Devices . . . 101
Approaches in Matrix Multiplication . . . 102
Compressed Sparse Formats . . . 103
Recent Hardware Architecture for Handling Sparsity . . . 104
Architectures for Power-Gating-Based Active Leakage Control . . . 111
Overview of Power-Gating . . . 111
Challenges and Trade-Offs in Power-Gating . . . 112
Power-Gating Efficiency Learner . . . 115
Self-Adaptive Power-Gating Architecture . . . 116
Test Chip and Measurement Results . . . 117
Conclusion and Future Roadmap . . . 120
References . . . 121

A. R. Trivedi (✉)
University of Illinois at Chicago, Chicago, IL, USA
e-mail: amitrt@uic.edu

J. Kung
Daegu Gyeongbuk Institute of Science and Technology (DGIST), Daegu, Republic of Korea
e-mail: jhkung@dgist.ac.kr

J. H. Ko
Sungkyunkwan University (SKKU), Suwon, Republic of Korea
e-mail: jonghwan.ko@gmail.com

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_9

Abstract

Artificial intelligence (AI) and machine learning (ML)-based decision-making is proliferating to application spaces with dynamic and evolving inputs such as Internet of things (IoTs). The need for real-time decision-making in such applications requires the edge devices in IoT networks to possess in situ intelligence processing capability. Edge intelligence in the networks is critical to avert unpredictable latency of an otherwise cloud-based intelligence processing. Edge intelligence in IoTs also minimizes their energy demand by avoiding raw data transmission and better preserving data privacy by only transmitting actionable information. Meanwhile, due to form factor and cost constraints and battery-powered operation, the energy budget and computing/storage resources for edge intelligence are very limited in a typical IoT node. Addressing such computational challenges in IoTs, in this chapter, an architectural framework for *self-powered edge intelligence* is reviewed. First, architectural techniques are reviewed to exploit sensors in IoTs to harvest energy from their environment to sustain local intelligence processing. Next, architectures that can identify and focus on regions of interest (ROI) are discussed to exploit sparsity in input and to minimize edge intelligence workload. Finally, learning-based architectures are discussed to reduce power wastage, such as due to leakage power. With a synergistic integration of the above architectural techniques, many IoTs can leverage self-powered edge intelligence to heighten awareness of their application domains.

Keywords

Energy scavenging · Image sensors · Power-gating · Region of interest (ROI) identification · Low-power computing · Reconfigurable computing

Evolution of Edge Intelligence and a Pathway to Self-Powered Intelligent Computations

Artificial intelligence (AI) and machine learning (ML) algorithms have shown that growing volume and variety of data, efficient computing and storage resources, and data-driven learning frameworks can be exploited for highly accurate predictions in many complex problems such as computer vision and natural language processing. The first-generation AI/ML algorithms were mostly employed on applications where prediction accuracy mattered the most, and improving computational efficiency was an afterthought. This has changed in the present applications where AI/ML platforms must simultaneously meet stringent accuracy, speed, and energy constraints in intelligence processing. Among the emerging AI/ML applications, Internet of things (IoTs) especially offer intriguing prospects. By augmenting distributed perception and control of IoTs with the data-driven learning of AI/ML, intelligence IoT devices can heighten awareness and unprecedented control of their application spaces. For example, in precision agriculture, a distributed camera

network can detect disease onset by classifying crop images as healthy or diseased to maximize the farm yield. Since network connectivity in remote agriculture fields can be unpredictable, intelligent IoTs reduce their reliance on the cloud nodes by possessing on-sensor intelligence. Similarly, IoTs with edge intelligence in a smart office can personalize workspaces without transmitting personal data to the cloud.

Figure 1 shows the hypothesized evolution stages of edge intelligence, similar to Zhou et al. (2019), but from a hardware perspective. At the first level of edge intelligence, the training of intelligence models is performed only in the cloud nodes, whereas the cloud and edge devices collaborate for inference. Especially, at this level of edge intelligence, edge devices are utilized to locally extract and only transmit the actionable information, thereby reducing the necessary communication bandwidth demand between the edge and cloud. Applications at this level of edge intelligence include keyword spotting in smart home devices such as Google Home (Google Home) and activity recognition in smart cameras such as Blink (Amazon Blink). By locally identifying the actionable inputs, a cloud node need not continuously receive data from the edges, such as from Alexa or Ring, but it is invoked only when an action may be desired. At the second level of edge intelligence, even though the training of intelligence models is restricted to the cloud, edge nodes perform end-to-end inference. Applications at this level include autonomous navigation of small drones in remote environments of limited cloud bandwidth (Shukla et al. 2021). An end-to-end edge intelligence is needed to operate on dynamically evolving inputs where a latency in the actions may lead to fatal consequences such as drone collision. Since latency from the cloud-based inference in remote environments may be high due to lower bandwidth, or at worse unpredictable, an end-to-end in situ edge intelligence is necessary. At the third level of edge intelligence, both training and inference capabilities locally within the edge device are needed. Although the edge device may inherit a cloud-trained initial intelligence model, it must update the model locally to adapt against application surroundings. Applications at this level include continuous learning or reinforcement learning in edge devices where the devices continuously update their intelligence model by interacting with the environment.

Edge Intelligence	Computing Constraints	Key Applications
Level 1	. Training on the cloud node. . Collaborative inference between edge and cloud nodes.	Keyword-spotting in smart home devices. Activity detection in smart cameras. Robotic vacuum cleaners.
Level 2	. Training on the cloud node. . End-to-end inference at the edge node.	Autonomous drones in remote application spaces such as forest and agriculture fields. Military surveillance devices in adversarial spaces.
Level 3	. Continuous learning and end-to-end inference at the edge.	Reinforced learning platforms to train and act by interacting with the environment. Continual learning.

Fig. 1 Evolution of edge intelligence. At level 1, edge devices collaborate with cloud for inference. At level 2, edge devices perform end-to-end inference. At level 3, edge devices perform both training and inference locally

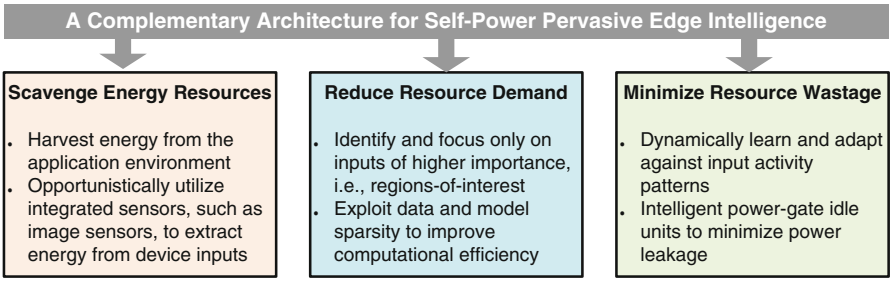


Fig. 2 A pathway for self-powered edge intelligence. In this chapter, complementary techniques on energy scavenging, computational energy efficiency, and minimization of energy leakage are reviewed for pervasive edge intelligence

Despite the intriguing prospects of edge intelligence in IoTs, most edge nodes are constrained in area and energy, limiting their budget for on-edge intelligence capabilities. Subsequently, emerging architectural techniques are reviewed to address this challenge from complementary viewpoints (Fig. 2). First, in section "Architectures for Energy Harvesting in IoT Edges", architectures to enhance the energy budget for on-edge intelligence by harvesting energy from the environment are reviewed. Specifically, techniques to leverage IoT sensors for opportunistically scavenge energy are discussed when the sensor inputs need not be processed. Second, in section "ROI-Aware Image Processing Architecture", architectures that can identify and focus upon regions of interest (ROI) are reviewed. By focusing only on ROIs, the computational efficiency of edge intelligence can improve dramatically. In section "Architectural Support for Handling Sparsity in IoT Devices", architectures that can exploit sparsity in input and parametric models for intelligence processing are reviewed. Since perception domains and computing models for IoT edge devices are often sparse, sparsity-aware computations in this section will minimize the computing and storage resource demand for on-edge intelligence. Readers can also refer to chapters in this book on approximate computing and subthreshold computing which are complementary techniques to our approach to enhance computational energy efficiency in this chapter. Additionally, since IoT edge nodes have low activity, in section "Architectures for Power-Gating-Based Active Leakage Control", learning-based architectures are reviewed to learn and adapt against varying application activities and environmental conditions to minimize power wastage. Synergistic integration of architectural techniques on energy harvesting, efficient workload processing, and efficient energy resource utilization will lay the foundations of self-sustained edge intelligence for next-generation intelligence IoTs.

Architectures for Energy Harvesting in IoT Edges

In IoT applications, wireless image sensor nodes are generally deployed in areas where human intervention for battery replacement is a costly operation (Law et al.

2011). Therefore, sensor nodes are expected to operate for an extended period with limited energy sources. A longer lifetime can be achieved by harvesting ambient energy in the environment (Cevik et al. 2015). However, energy harvesting generally requires additional devices (thermoelectric, piezoelectric, photovoltaic, etc.). An alternative approach to a self-powered sensor node can be using the sensor array itself as an energy harvesting device. Since the pixel array is used for sensing only for a limited fraction of time, it can be configured to harvest energy during idle time and store harvested energy in a battery or supercapacitor. While a few studies have shown the feasibility of using an image pixel array for harvesting (Law et al. 2011; Kim et al. 2014; Wang and Leon-Salas 2015; Nayar et al. 2015; Chiou and Hsieh 2015), these studies only considered powering a pixel array and peripherals.

A Self-Powered Image Sensor System with Autonomous Mode Management (AMM)

As an important goal of a sensor node is to deliver the visual information, it is generally integrated with an image processor, memory, transmission controller, and power management unit (PMU) (Fig. 3). A recent study Ko et al. presented a single-chip system to show the feasibility of a self-powered image sensor system. The mixed-signal SoC integrates an energy harvesting pixel array with a multi-output PMU, a digital processing engine with an on-chip SRAM for moving object detection, and a transmission controller. The system incorporates a reconfigurable CMOS active pixel sensor (APS) array that operates as a photodetector while sensing/processing a frame and as a photovoltaic cell in between successive frames to harvest energy. The CMOS sensor is coupled with an on-chip PMU that multiplexes a single inductor for boosting the harvested input for intermediate energy storage and generating three regulated output voltage domains. An image processing engine with an on-chip SRAM identifies the region with moving objects to reduce transmission energy with reliable information delivery.

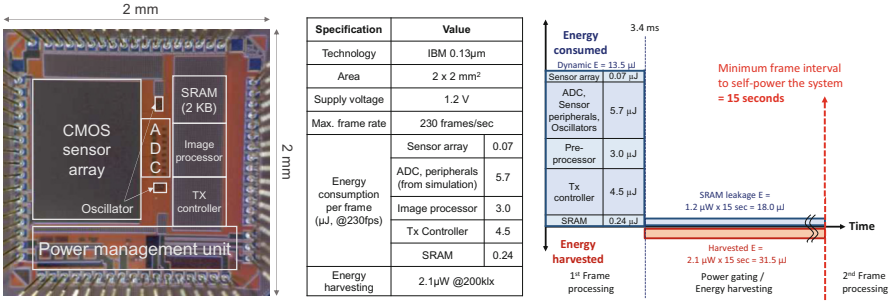


Fig. 3 (**a**) Die photo of the sensor node, (**b**) key performance parameters of the system, (**c**) and a diagram showing the consumed/harvested energy over time

An AMM unit in the system controls the switching between the sensor's imaging and harvesting mode and transitions between the regulators' boost and buck mode. The mode switching can be externally controlled or autonomously managed based on available stored energy. In energy-autonomous imaging mode, the mode switching signal is self-generated by the system. Such a decision is made by sensing the voltage drop in the energy storage and assessing how much energy is required to process the next frame. If the energy level in the storage is below that minimum limit, the system decides to harvest before allowing the next frame to capture. Thus, in the self-powered case, the frame rate becomes a system-defined variable and varies depending on available energy. In practical operations, the demanded frame rate can push the system into sense, but if enough energy is not available, the AMM will stop sensing and go to harvesting mode.

Factors Affecting Self-Power Performance

A test chip is designed in a 0.13 μm CMOS technology node, as depicted in Fig. 3a. It can process image frames with 128 × 96 pixels at the maximum frame rate of 230 frames/s. The design demonstrates the peak harvested power of 2.1 μW at the sensor array's output. Based on the peak harvested power and the measured power dissipation of the different components, the sensor can be self-powered while processing a frame every 15 s (Fig. 3c). The maximum frame rate that can be supported by energy harvesting is affected by various system factors. The factors include pipelining architecture, SRAM supply voltage, pixel size, and power converter efficiency.

Effects of a Processing Pipeline

The self-power performance is affected by image processing energy. To minimize the processing energy of the system, the size of the buffer between the image sensor, processor, and transmission controller should be minimized. Therefore, instead of frame-level pipelining, a block-level pipelining scheme can be applied to have only block-sized buffers between the components. A block-wise readout enables the block-level pipelining at the CMOS sensor instead of the conventional row-wise readout. Also, the processing engine can be operated at the highest voltage and frequency, instead of a low-voltage operation, to enhance the idle periods between frames and, hence, harvested energy. The digital logic can be powered down in between frames by controlling the PMU output to reduce leakage energy (frame-level power-gating).

Effects of Unit Pixel Size

The self-powered frame rate is limited by harvested power from the sensor array. One of the reasons is extra metal layers at the pixel boundaries. These metal layers are placed because of the foundry's minimum metal density restrictions, causing a decrease in the amount of pixel area exposed to light. Therefore, increasing unit pixel area can increase light exposure, thus enhancing energy harvesting

performance. Increased pixel area is also expected to improve the dynamic range of image sensing. The major disadvantage of increasing unit pixel size is the reduced image resolution per array area. To keep the same number of pixels (resolution), more sensor array area will be needed. Similarly, to keep the array area the same, the number of pixels in the array must be reduced. While lower image resolution results in lower perceptual quality to the users, it has an advantage in system energy consumption because it reduces the computation energy and transmission energy per frame, enhancing the self-powered frame rate.

Effects of SRAM Leakage Energy

The system's self-power performance can be enhanced by reducing standby energy dissipation in between frames and processing (active mode) power demand. One way to reduce the SRAM leakage power is to operate it at a lower supply voltage. Once the minimum voltage at which SRAM can function and/or retain data has been identified, supply voltage can be reduced to the minimum voltage during the idle mode. To enable the adaptive supply control, the SRAM power supply should be separated from the logic engine.

Effects of Power Converter Efficiency

The self-power performance is closely related to the efficiency of the power converter. The boost converter of the system presented in Fig. 3 showed 24% efficiency at an input source power of 2.76 μW (simulated considering all parasitics). This low efficiency is mainly due to switching losses in the power stage MOSFETs. Since the power converter uses only one power stage, it cannot be optimized simultaneously for two widely varying power ranges (μW range during boost, mW range during buck), thus leading to the degraded efficiency for low input power. A possible solution for this would be to split up the power converter into boost and buck stages – this would allow us to improve efficiency by decoupling the buck and boost stages and optimally size them to obtain higher efficiency in their typical operating ranges.

ROI-Aware Image Processing Architecture

Conventional image sensor systems have focused on the capabilities of wireless sensor nodes to capture, process, and transmit visual information to the base station while leaving the task of video analysis to human operators (Hampapur et al. 2003). In this configuration, delivering images with higher perceptual quality to human operators is critical. Therefore, the sensor node design requires exploring resource-quality scalability to allocate limited resources for better visual information optimally.

As human visual attention focuses on the ROI in an image, ROI-based processing (ROI detection and ROI coding) is a common design approach for better energy-quality scalability of sensor nodes (Grois and Hadar 2014). In surveillance applications where the ROIs are defined as the regions with moving objects, sensor

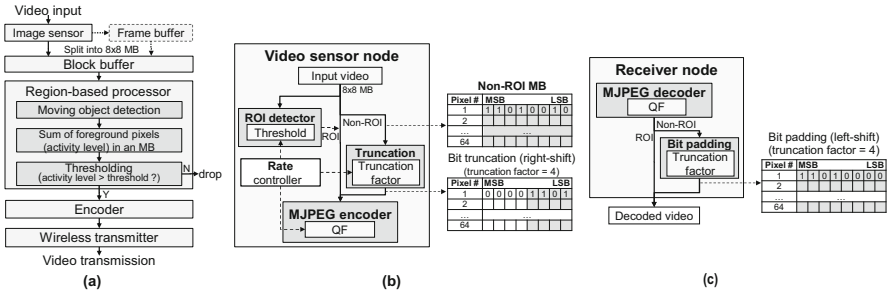


Fig. 4 (**a**) A wireless image sensor platform with a block-wise region-based processing model (**b**) proposed bit-truncation method

systems usually incorporate moving object detection methods to optimally allocate the resources while preserving the quality of moving objects (Fig. 4a). Once the ROI is determined, ROI coding allocates the available data rate to the ROI/non-ROI for higher ROI quality and graceful degradation of the non-ROI quality (Meddeb et al. 2014).

The optimization of video quality and energy consumption also involves variation in a wireless channel condition. An optimal energy-quality trade-off under varying channel conditions requires system-wide feedback control that adaptively tunes the target data rate. Once the target data rate is determined, the remaining challenge is to guarantee the data volume generated by an image processor match the target data rate. The mismatch imposes energy/area overhead due to large buffer requirements.

Moving Object Detection Architecture

When the ROI is defined as a region with moving objects, detection of the ROI can be divided into two categories depending on the complexity and robustness: low-power moving object detection and noise-robust moving object detection.

Low-Power Moving Object Detection

Several prior studies have proposed moving object detection methods for resource-constrained applications using more straightforward approaches based on frame differencing (Chefi et al. 2013) and edge detection (Imran et al. 2013). Alternatively, approaches based on a combination of ED and FD have been suggested; Kim et al. proposed an algorithm based on the edge map of the inter-frame difference image (FD+ED) (Kim and Hwang 2002), and Ko et al. suggested an algorithm based on the inter-frame difference of edge maps (ED+FD) (Ko et al. 2015). However, these low-overhead methods are susceptible to noise in the dynamic environment and are not suitable for outdoor sensor platforms.

Noise-Robust Moving Object Detection

One of the challenges in image sensing from the sensor node is random noise induced from the image sensor array, corrupting the captured images. In addition to the random noise, in their outdoor applications, the sensor platforms are often exposed to a dynamic environment, where objects of interest usually move amidst noisy backgrounds, for example, snow, rain, etc. (Brutzer et al. 2011). A common approach for noise-robust moving object detection is background subtraction with a multimodal background model. The most widely used modeling method is the Gaussian mixture model (GMM) (McKenna et al. 1999) which employs a weighted sum of Gaussian distributions to describe the probability of observing the intensity at each pixel. Other multimodal background modeling approaches have been proposed based on the codebook (Kim et al. 2005), kernel density estimation (KDE) (Elgammal et al. 2000), and eigenspace (Oliver et al. 2000). Another way of identifying moving objects is to calculate optical flow (OF), the changes of motion between frames with the assumption of constant brightness. While providing robust detection performance in a noisy environment, a critical drawback of these methods is substantial computation and memory requirements for storing the background model or computing the velocity of each pixel (Benezeth et al. 2010).

A recent study Ko and Mukhopadhyay (2016) presented a low-power noise-robust moving object detection for resource-constrained sensor platforms. The essential contribution is designing a block-level rank closing operation to improve noise robustness of the existing moving object detection method using sequential edge detection and frame differencing (ED+FD). The approach provides comparable performance with the existing noise-robust method based on GMM but consumes 28X and 2.7X lower area and processing energy, respectively. The ASIC (130 nm CMOS) realization shows only 2.1% energy overhead compared to the whole system; however, the system-level analysis shows significantly lower energy at the same quality of ROI. The primary advantage comes from the significant reduction in the on-chip memory capacity/energy.

ROI-Based Coding Architecture

Once the ROI is detected, a video can be processed more efficiently by focusing on the ROI. The ROI-based coding methods can be divided into temporal and spatial methods.

Temporal ROI-Based Coding

The simplest approach to ROI-based coding is to drop non-ROI blocks and encode/transmit only the ROI blocks (Tuan and Chen 2015). This approach reduces encoder energy and overall data volume as non-ROI blocks are not encoded or transmitted. However, it can suffer from the loss of context information and lower overall visual quality since no visual information on the background is delivered. Moreover, in case of the false negatives, i.e., the ROI is falsely determined as the non-ROI, the ROI quality degrades significantly since no ROI information is

transmitted. To address these drawbacks, Lai et al. (2004) have proposed the multi-rate approach that transmits non-ROI blocks with a frame rate lower than that of ROI blocks. However, whenever the frames with non-ROI blocks are transmitted, the transmit volume increases significantly, requiring a large buffer to accommodate high fluctuation in the encoding rate. Moreover, for correct reconstruction of the image frames without non-ROI blocks, a sensor node needs to transmit block identifiers that contain the location (or sequence number) of the blocks.

Spatial ROI-Based Coding

Spatial ROI-based coding approach is to transmit both ROI and non-ROI blocks but compress non-ROI blocks more than ROI blocks. One of the spatial techniques is to use multiple QF values in a frame: higher QF for ROI blocks and lower QF for non-ROI blocks (Hu et al. 2012). However, the multi-QF approach also requires extra transmission of one additional QF value and the ROI map indicating whether a block is encoded using the higher or lower QF. Also, as the standard MJPEG uses a single QF in a frame, the multi-QF approach adds energy/area overhead to the MJPEG encoder/decoder to enable frame encoding with the two different QFs. Another spatial approach is to preprocess non-ROI blocks to enable higher compression at the encoder. The prior studies have proposed pre-filtering of non-ROI blocks via low-pass filters such as a median filter (Tsapatsoulis et al. 2007) and a Gaussian filter (Grois et al. 2011), which reduce high-frequency information in non-ROI blocks, enabling high compression with the same QF. Pre-filtering can be an attractive solution because the unit operation of median or Gaussian filtering is low complexity. However, it is an inefficient solution for online tuning of non-ROI size/quality since a larger filter size for more smoothening (more compression) requires heavy computation, increasing computation energy (Grois and Hadar 2014).

One recent study Ko et al. (2016) proposed a tunable and low-complexity ROI-based data rate control scheme. As Fig. 4b, c shows after the ROI decision unit classifies ROI/non-ROI MBs, low-order bits in 64 pixels of non-ROI MBs are truncated and encoded by the MJPEG encoder, while ROI MBs are encoded without truncation. After being transmitted to the receiver node, the truncated non-ROI MBs are left-shifted by the same number of bits as in the truncation operation at the sensor node. The hardware design of the truncation method is simple because it is based on a bit-shift procedure. Also, the truncation level can be easily tuned by controlling the number of shift operations. Therefore, the bit-truncation method can be effectively integrated with an online rate controller.

Resource-Aware Control of Target Data Rate

Conventional Target Data Rate Control

As the channel condition worsens, a transmitter generally enhances channel reliability at the cost of a lower channel data rate or higher signal power (Fallah et al. 2008). This conventional link adaptation technique, which is the independent

control scheme, may result in significant quality degradation when the channel's data rate is lower than that of the encoder. To address this problem, Haratcherev et al. proposed cross-layer signaling, which informs the encoder of the channel data rate so that the encoder can change its data rate as well (feedback control scheme) (Haratcherev and Taal 2005). However, suppose the feedback control scheme uses conventional rate-controlled encoders. In that case, it controls the source data rate by changing only the quality of the entire video, which may result in a low-quality ROI (content-unaware feedback control scheme). Although the feedback controller with existing ROI-based processing approaches (i.e., the content-aware and energy-unaware feedback control scheme) can optimize the quality of the ROI, it is subject to an energy increase in a channel rate decrease or a signal power increase. Therefore, achieving an optimal system-level energy-quality trade-off under varying channel conditions requires system-wide feedback control that adaptively tunes parameters of an encoder and a transmitter.

Energy- and Content-Aware Target Data Rate Control

Energy- and content-aware control integrates ROI-based processing and the optimization of transmission energy under the variation of wireless channel conditions. The main idea is to reduce the source rate (R_S) according to a transmission power (P_{TX}) increase or a channel rate (R_C) decrease, while the existing control schemes decrease the R_S only when it becomes larger than the R_C. The quality degradation of the ROI, due to the reduced R_S, can be minimized by using a content-aware control scheme. The content awareness enhances the energy-aware control since it better preserves the quality of the ROI at the same source data rate by dropping non-ROI MBs and increasing the quality of ROI MBs. The reduced number of encoded MBs leads to (i) a decrease in the source data rate (like other works) and (ii) a reduction of computation energy at the encoder.

The characteristics of each control scheme are summarized in Table 1. By exploiting both energy and content awareness, the energy- and content-aware

Table 1 Summary of target rate control schemes

Control scheme	Feedback	Awareness		Source rate reduction	Parameters for source rate control
		Energy	Content		
Independent control	No	No	No	No source rate control	
Energy and content-unaware control	Yes	No	No	When $R_C < R_S$	Quality of all MBs
Energy-unaware content-aware control	Yes	No	Yes	When $R_C < R_S$	Quality or priority of each MB
Energy-aware content -unaware control	Yes	Yes	No	When $P_{TX} \uparrow$, $R_C \downarrow$	Quality of all MBs
Energy- and content-aware control	Yes	Yes	Yes	When $P_{TX} \uparrow$ $R_C \downarrow$	# MBs to encode, quality of encoded MBs

control scheme optimizes the system performance in two ways. First, the controller guarantees bounded transmission energy and quality distortion due to a wireless channel by reducing the source data rate according to the transmission parameters satisfying the BER target. Also, source rate control using both the number of MBs to encode and the quality of these MBs further optimizes the ROI and the energy consumption in the computation.

Resource-Aware Control of Encoding Data Rate

Challenges in Data Rate Control

Once the target data rate is determined, the data rate generated by an image processor should be matched with the target data rate. If the encoder generates too much data that the transmitter cannot accommodate, the data should be buffered to avoid random packet drop at the transmitter. However, buffering introduces variable latency between the source and destination and imposes energy/area overhead due to large memory requirements. The challenge is that the encoding rate can vary due to the variable content of video frames. The wireless channel bandwidth and the maximum transmission data rate can also change over time. Consequently, there is a need for an online rate controller that matches the encoding rate with the transmission data rate.

The recent encoders such as H.264/AVC incorporate rate controller schemes that allocate proper bit budgets and determine a quantization parameter (QP) to minimize quality degradation based on rate-distortion optimization (Fig. 5a). However, calculation and update of the parameters require complex computation with appreciable energy cost, and hence it is not suitable for energy-constrained systems (Huang and Lin 2009). On the other hand, the existing ROI-based coding schemes use lower-complexity codecs such as motion JPEG (MJPEG); however, they do not employ rate controllers to dynamically modulate the ROI-based coding parameters. Therefore, an ROI-based coding scheme with a low-complexity rate controller is necessary to improve the quality of visual information under stringent system energy constraints.

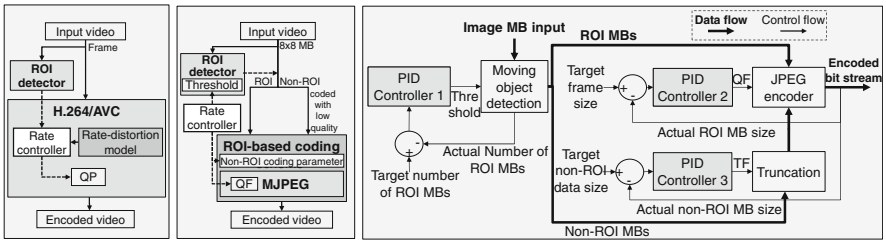


Fig. 5 ROI-based rate controller design (**a**) tied to H.264/AVC and (**b**) based on a simple encoder (MJPEG). (**c**) Diagram of the low-power online rate controller

Low-Power Data Rate Control

To design a rate controller with less complexity, a recent study Ko et al. (2016) adopted a proportional-integral-derivative (PID) controller. The system comprises two control parameters (the threshold and the QF), one variable (the source data rate), and three PID controllers (Fig. 5b, c). The first PID controller modifies the threshold so that it can maintain the target number of encoded MBs. The second PID controller controls the QF to maintain the frame size, and the truncation factor is controlled by the third PID controller to maintain the non-ROI data size per frame.

The low-power controller consumes significantly lower system energy at a given target rate than H.264/AVC due to the reduced computation overhead. Although it requires more data transmission to achieve the same quality, with its significantly low computation energy and reduced buffer overhead, it consumes less system energy than other approaches at the same target ROI quality.

Architectural Support for Handling Sparsity in IoT Devices

To understand the underlying system or preprocess external signals, generalized matrix multiplication (GEMM) is extensively used, i.e., a standard algorithm in linear algebra and statistics. The applications of GEMM operations include, but not limited to, data science (Bennett and Lanning 2007), ML (Anandkumar et al. 2014), graph analytics (Mattson et al. 2013), image processing (Vasudevan et al. 2017), traffic control (Dunne and Potts 1964), motion planning (Gao et al. 2017), and recommendation system (Naumov et al. 2019). As the computing power of microprocessors in IoT devices improves, more data are collected and processed in a small form factor. Besides, as the size of sensors minimizes and the resolution of sensed data increases, it becomes essential to process and fuse various types of data in mobile/IoT devices. Here, the IoT devices are pieces of hardware, such as sensors, appliances, machines, or robots, that are programmed for specific tasks and can communicate with each other over the Internet or different types of wireless networks.

In real-world applications, the matrix that defines the underlying system may be sparse for many reasons (Grasedyck et al. 2013; Hapla et al. 2013; Ordejon 1998; Han et al. 2016a). Also, many scientific problems require sparse matrix multiplication (SpGEMM) which becomes the performance bottleneck due to irregular data accesses and core underutilization. As zeroes in a sparse matrix consume memory bandwidth and occupy processing engines without meaningful computations, compressed storage formats and their corresponding hardware architectures have been proposed and developed (Bank and Douglas 1993; Robinson and Cherry 1967; Qin et al. 2020; Kung et al. 2019; Pal et al. 2018; Zhang et al. 2020). With the use of compressed storage format, however, index matching overhead and load imbalance problems are revealed. Thus, it is crucial to design a storage format carefully and its hardware architecture depending on the range of sparsity levels under consideration.

Approaches in Matrix Multiplication

For GEMM operations, two input matrices **A** and **B** are provided and the output matrix **Y** is generated. In computing the output matrix **Y**, two approaches can be used: (i) inner product approach and (ii) outer product approach. Each approach has its advantages and disadvantages; thus, proper selection is needed depending on the sparsity level and on-chip memory size.

Inner Product-Based Approach

In many cases, the inner product approach is selected for the matrix multiplication as it is a normal process in dense GEMM operations. It involves multiple dot products between row vectors of matrix $\mathbf{A} \in \mathbb{R}^{N \times K}$ and column vectors of matrix $\mathbf{B} \in \mathbb{R}^{K \times N}$ (Fig. 6a). Each element of matrix $\mathbf{Y} \in \mathbb{R}^{N \times N}$ is computed by

$$y_{ij} = \sum_{k=0}^{K-1} a_{ik} \cdot b_{kj}, \tag{1}$$

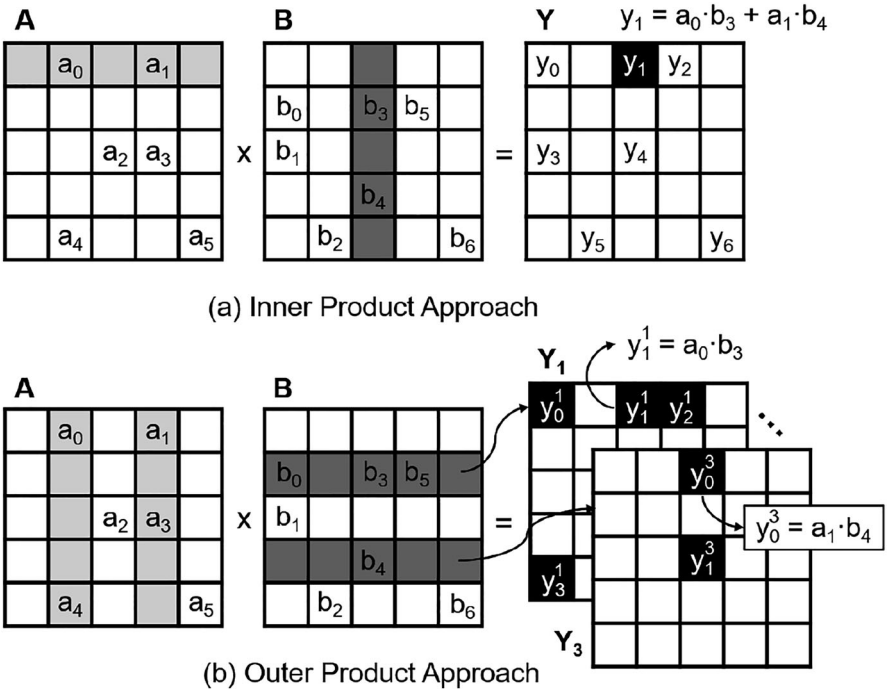


Fig. 6 Two approaches for matrix multiplication: (**a**) inner product approach and (**b**) outer product approach

where (i, k), (k, j), and (i, j) are coordinates of an element in matrix $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{Y}$, respectively. The required number of multiply-accumulate (MAC) operations for computing $\mathbf{Y}$ becomes $K \times N^2$. Note that the addition of $a_{ik} \cdot b_{kj}$ terms happens right after the multiplication is performed. Thus, the inner product approach shows high output reuse (or partial sum reuse). However, the inner product approach has low input reuse for one of two input matrices, e.g., a row vector of $\mathbf{A}$ is stationary to the processing engines, while a column vector of $\mathbf{B}$ changes for each dot product. When the GEMM operation becomes sparse, it becomes a challenge to match the index of a column in $\mathbf{a}_i$ and a row in $\mathbf{b}_j$, i.e., index "k" in Eq. (1).

Outer Product-Based Approach

To provide high input reuse by sacrificing the level of output reuse, the outer product approach can be used. For SpGEMM operations, the outer product approach removes the overhead of index matching. To compute a matrix multiplication in the outer product approach, a set of multiplications is performed between element pairs of columns of $\mathbf{A} \in \mathbb{R}^{N \times K}$ and rows of $\mathbf{B} \in \mathbb{R}^{K \times N}$. Then, "$N$" different partial products ($\mathbf{Y}_k$'s) are summed together to obtain the final $\mathbf{Y}$ (Fig. 6b):

$$\mathbf{Y} = \sum_{k=0}^{K-1} \mathbf{Y}_k = \sum_{k=0}^{K-1} \mathbf{a}_k \otimes \mathbf{b}_k, \tag{2}$$

where "k" is index of each partial product, $\mathbf{a}_k$ is the kth column vector of $\mathbf{A}$, and $\mathbf{b}_k$ is the kth row vector of $\mathbf{B}$. Unlike the inner product approach, the addition of partial products occurs after all multiplications are completed. Thus, the outer product approach shows high input reuse but poor output reuse as all $\mathbf{Y}_k$'s are kept in on-chip or off-chip memory for the reduced operation. As shown in Fig. 6, to obtain the same output y_1 in the inner product approach, $\mathbf{Y}_1$ and $\mathbf{Y}_3$ must be added together to produce "$y_1 = y_1^1 + y_0^3 = a_0 \cdot b_3 + a_1 \cdot b_4$".

Compressed Sparse Formats

As zeroes dominate in a given matrix, i.e., the sparsity of the matrix increases, fetching the entire matrix in the original dense format significantly reduces the effective memory bandwidth. To utilize the memory bandwidth with useful data, various types of sparse data format have been proposed (Bank and Douglas 1993; Robinson and Cherry 1967; Qin et al. 2020; Kung et al. 2019). The coordinate list stores a list of {row, column, value} tuples where the data is typically stored in row-major order. The size of metadata to locate nonzero values in the coordinate list format is large as it needs to store both row and column coordinates. To reduce the metadata size of indexing either rows or columns, compressed sparse row (CSR) or compressed sparse column (CSC) is used (Eisenstat et al. 1984). In the CSR format, rows' extents (row pointers) are stored instead of row coordinates. The row pointer identifies the number of nonzero values in that row. The CSC format is similar to CSR format but instead storing row indices and the extents of columns (column pointers).

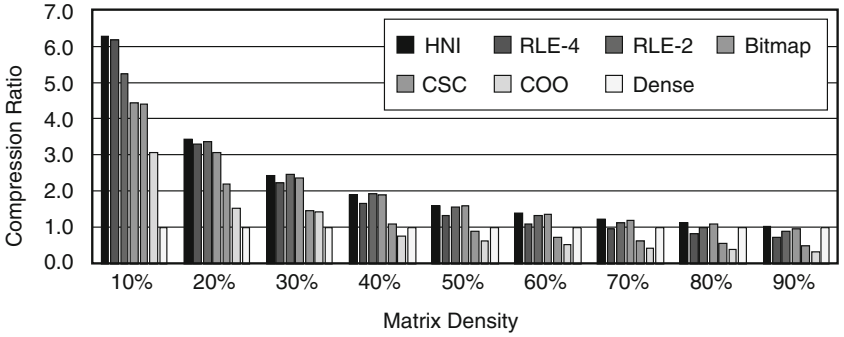


Fig. 7 The resulting compression ratio using various sparse matrix formats depending on the sparsity of a 2048 × 2048 weight matrix

There is another set of sparse matrix formats that does not explicitly store coordinates of nonzero values. In run-length encoding (RLE), consecutive zeroes between two nonzero values are clustered together, and the zero count is stored as an indicator with a predefined bit-width, e.g., 2 bit or 4 bit (Robinson and Cherry 1967). For instance, with a 4-bit run-length code, up to 15 zeroes per decoding cycle can be skipped. The simplest way of identifying nonzero values is by storing a bitmap with marking 1 for nonzero values (Qin et al. 2020). The size of the bitmap, however, remains the same even at a high sparsity level. The compression ratio can be improved by applying the Huffman coding on the bitmap, i.e., Huffman-coded nonzero indication (HNI) (Kung et al. 2019).

The compression ratios using different sparse matrix formats are compared in Fig. 7. The simulated matrix size is 2048 × 2048 at varying sparsity levels. For sparse formats, the data includes nonzero values as well as the metadata for their coordination. Thus, it is beneficial to use a sparse matrix format when the matrix density is less than 70%, i.e., sparsity above 30%. At low sparsity levels, i.e., sparsity below 50%, using the coordinate list or CSC/CSR format is still worse than using the dense format. The HNI or Bitmap shows the highest compression ratio at sparsity from 20% to 50%. When the matrix becomes highly sparse, i.e., above 80%, the RLE-4 format becomes as efficient as the HNI format. Note that the HNI format shows the highest compression ratio at any sparsity levels. Reducing the total data size as much as possible is important as accessing data at memory blocks consumes significantly higher energy compared to arithmetic units (Horowitz). For example, accessing 32-bit data from 32-KB SRAM consumes 50× higher energy compared to adding two 32-bit integers in 45-nm CMOS technology (Table 2).

Recent Hardware Architecture for Handling Sparsity

Computing with sparse matrices puts a challenge in the design of processing units as it introduces the irregular data access pattern and index matching overhead.

Table 2 The Energy Consumption on Different Operations in 45 nm CMOS Technology (Horowitz)

Operation	Energy [pJ]	Relative cost
32-bit ADD (int)	0.1	1
32-bit ADD (float)	0.9	9
32-bit MULT (int)	3.1	31
32-bit MULT (float)	3.7	37
32-bit SRAM Access (32 KB)	5.0	50
32-bit DRAM Access	640	6400

Recently, a collection of research work proposed efficient hardware architectures for handling sparsity (Qin et al. 2020; Kung et al. 2019; Zhang et al. 2020; Pal et al. 2018). Each work focuses on either of the two approaches explained in section "Approaches in Matrix Multiplication". Also, some of the prior works utilize one of the sparse matrix formats introduced in section "Compressed Sparse Formats".

Hardware Architecture for Inner Product Approach

When SpGEMM is performed by the inner product approach, a dot product is performed between a row vector **a** of matrix **A** and a column vector **b** of matrix **B**. When doing the dot product, it is necessary to multiply two elements that share the same index, called an *intersection*. In Han et al. (2016b), the CSC format is used to represent a sparse matrix, and the next nonzero value in the row vector **a** is detected by using *leading nonzero detection (LND)* unit in the processing core (Fig. 8). It only shows a single processing unit in Fig. 8. Multiple processing units are clustered as a group, e.g., four processing units, for improved parallelism and scalability. The LND unit looks at elements in a vector **a**, Act in Fig. 8, from neighboring processing units and then picks the one with the smallest index. Then, this index is used to address the corresponding column in matrix **B**. As only nonzero elements are stored in "sparse matrix SRAM (**B**)," the relative index is read, and it is used to identify the exact address of an output (partial sum) to be updated.

Besides using a less efficient CSC format, the access data size can be significantly reduced by using the HNI format (refer to Fig. 7). As the HNI format uses the Huffman coding, the hardware architecture proposed in Kung et al. (2019) includes the parallel Huffman decoders and lookup tables (LUTs) to store the Huffman trees (Fig. 9). In the proposed architecture, four MACs are grouped to update a single output or four various output elements depending on the mode of operation. To maximize the performance with high parallelism, a Huffman decoder is attached to each MAC unit. As shown in Fig. 9b, the decoder takes in a 4-bit sequence at a time for higher performance. If the symbol is not detected (symbol identifier = 1), then the Huffman LUT outputs a relative address to look at in the next decoding cycle. With the symbol detected (symbol identifier = 0), the decoder outputs a symbol with a predetermined bit-width (e.g., 8 bit). Depending on the number of ones in

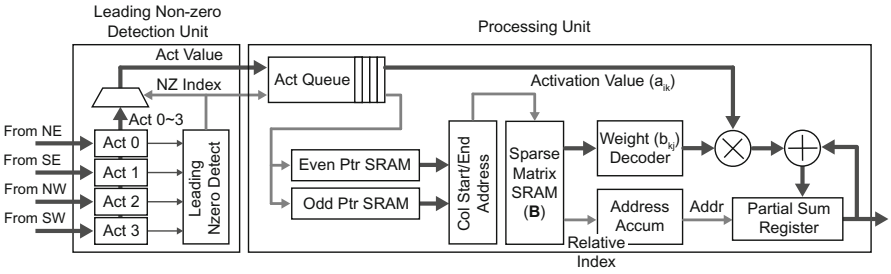


Fig. 8 The hardware architecture for SpGEMM operations using CSC format (Han et al. 2016b). It uses *LND* unit to skip zeroes in a row vector **a** of matrix **A**

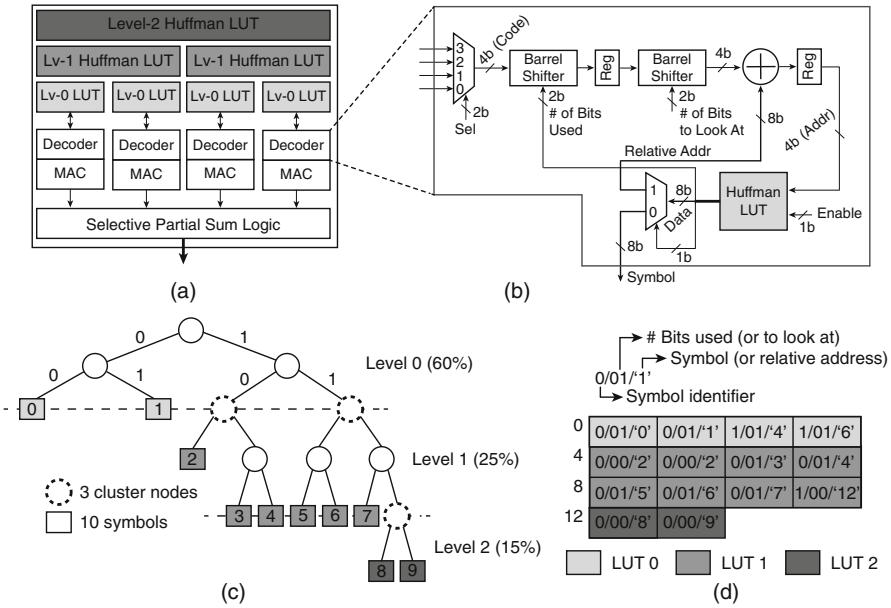


Fig. 9 The hardware accelerator for SpGEMM operations with HNI format (Kung et al. 2019). (**a**) The overall architecture of sparse processing engine with the hierarchical Huffman LUTs. (**b**) The design of a parallel Huffman decoder for real-time decoding. (**c**) The Huffman tree separated by multiple levels with depth = 2 and (**d**) its associated multilevel Huffman LUTs

the decoded symbol, a MAC unit becomes busy for that number of cycles. With the symbol "10010110," the corresponding MAC unit computes for the next four cycles. By using more efficient HNI format, the performance improves by 9.48∼27.57% on language modeling benchmarks compared to the hardware using CSC format (Kung et al. 2019).

Instead of finding the next intersection between two vectors at a time, ExTensor (Hegde et al. 2019) utilizes a tree representation to efficiently skip ineffectual coordinates. The hardware architecture to efficiently find intersections between

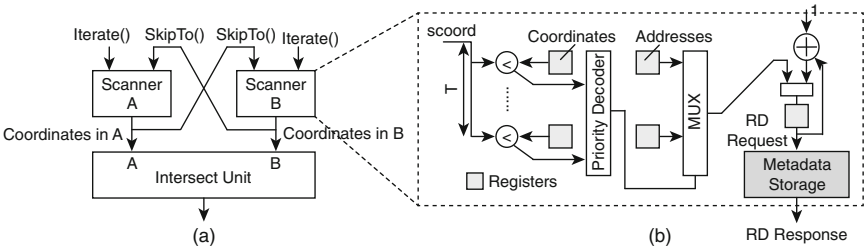


Fig. 10 (**a**) The hardware support for the bidirectional skipping mechanism for the optimized intersect architecture. (**b**) The scanner design, at which SkipTo() function is performed, is shown in detail

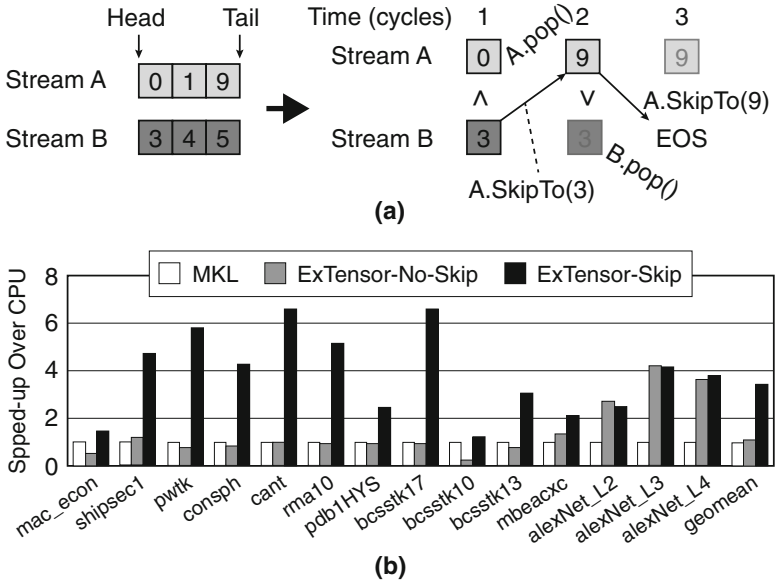


Fig. 11 The skip operation supported by ExTensor (Hegde et al. 2019). (**a**) An example of SkipTo() function calls. (**b**) The performance improvement by using ExTensor over Intel CPU with Math Kernel Library (MKL)

two coordinate streams A and B is shown in Fig. 10a. The scanner iterates over a stream of coordinates. Note that a high-dimensional tensor consists of many streams which are hierarchically intersected. To allow more efficient stream generation, SkipTo() function is implemented in a hardware module. The bidirectional skipping mechanism between scanners A and B allows a multistep jump as shown in Fig. 11a. An input coordinate (scoord) from another scanner is compared to "T" consecutive elements in the FIFO (Fig. 10b). The value "T" determines the efficiency of skipping mechanism. The intersect unit simply compares two coordinates from streams A and B and outputs the coordinate if they are identical, i.e., *intersection hit*. Otherwise,

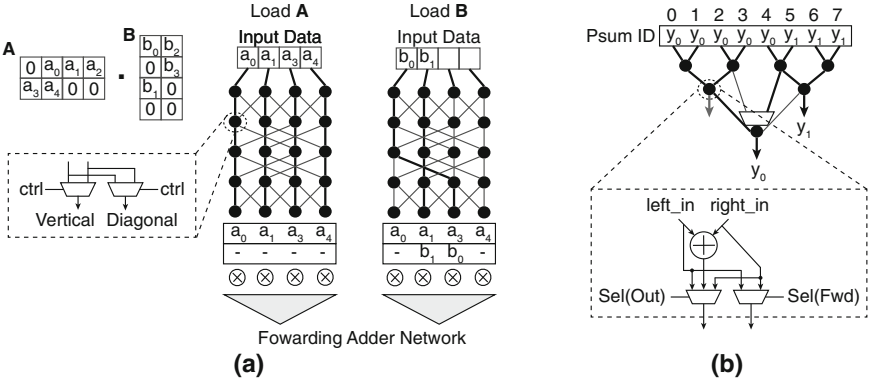


Fig. 12 The hardware architecture for processing irregular SpGEMM operations. (**a**) The overall architecture of SIGMA with Benes distribution network (Qin et al. 2020). (**b**) Forwarding adder network for the flexible output reduction

the intersect unit pops the smaller element at either FIFO A or B. Compared to computing SpGEMM on CPU with MKL support, ExTensor with the skipping mechanism improves the performance by 3.1× on average (Fig. 11b).

Recently, a two-dimensional systolic array is widely used for processing GEMM operations (Jouppi et al. 2017). However, the systolic array is not well suited at dealing with sparse and irregular matrices. To flexibly compute SpGEMM operations between two matrices with arbitrary shapes, SIGMA (Qin et al. 2020) presents a 2D processing array with a flexible distribution network and a programmable reduction network (Fig. 12). To design a flexible distribution network, SIGMA adopts a Benes network (Arora et al. 1990). The Benes network is a non-blocking multistage network allowing any source to connect with any destination without any contention (Fig. 12a). First, the consecutive nonzero data from matrix **A**, a_0, a_1, a_3, a_4, are loaded to the multiplicand buffer. Note that a_2 is not loaded as SIGMA identifies rows with all zeroes in matrix **B**, i.e., the fourth row in the example. Then, the corresponding nonzero values from matrix **B** are loaded to the multiplier buffer by programming the Benes network. For b_0, the path is programmed to be "V-V-D-V." For b_1, it is programmed to be "V-V-V-V." As the outputs from multipliers may have different destination indices, a forwarding adder network is used to reduce partial sums flexibly. The example in Fig. 12b shows how two different output values, y_0 and y_1, can be computed via the forwarding adder network.

Hardware Architecture for Outer Product Approach

Hardware accelerators for the inner product approach focus on finding matched indices between two sparse vectors. The stalls for waiting matched indices between highly sparse matrices may incur underutilization of processing elements. The outer product approach eliminates the overhead of index matching. In addition, it

maximizes the input data reuse by performing all necessary multiplications when two vectors are loaded. The issue with the outer product approach is in the merge phase that requires all intermediate partial products to be reduced to a single output matrix.

In OuterSPACE (Pal et al. 2018), a set of linked lists is used to store intermediate partial products. An example of the linked lists during the process of outer product-based matrix multiplication between matrices **A** and **B** is provided in Fig. 13a. The example assumes four processing elements in each processing tile (Fig. 14). Each PE multiplies one nonzero element from a column of **A** with all the nonzeroes in the matched row of **B**. The partial products are stored in a linked list pointed by a row pointer (Fig. 13b). The merge phase scans the linked lists pointed by the row pointers and adds them together when the column index matches. To do so, the

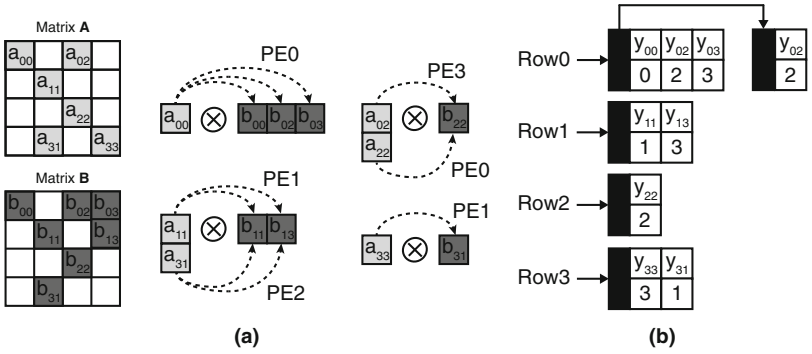


Fig. 13 (**a**) An example of outer product matrix multiplication and (**b**) the linked list representation of partial products used in OuterSPACE (Pal et al. 2018). In this example, four PEs are assumed where each PE multiplies one nonzero element from a column of **A** with all the nonzeroes in the matched row of **B** (compressed row mode)

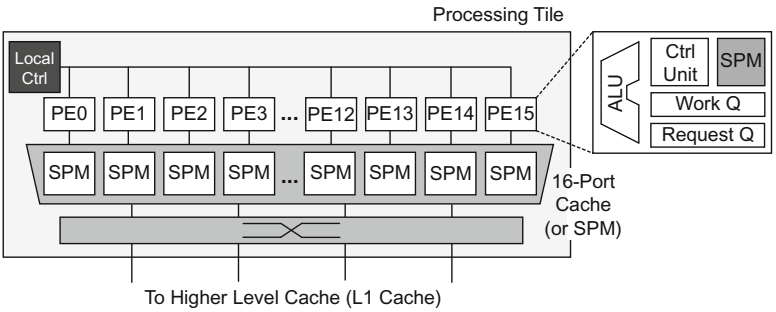


Fig. 14 The architecture of a processing tile in OuterSPACE accelerator (Pal et al. 2018). The processing tile consists of 16 processing elements with dedicated scratch pad memories

head of each row is fetched, and it is sorted by column index. Then the smallest indexed element from the list is stored in the memory location. This merge strategy in OuterSPACE focuses on minimizing the memory traffic. The crossbar attached to scratch pad memories is utilized to move data that need to be summed together. Also, it acts as a flexible router node to communicate with higher-level caches.

Yet, the outer product approach described in Fig. 13a shows poor output reuse as "K" partial product maps need to be merged for $\mathbf{A} \in \mathbb{R}^{N \times K}$ and $\mathbf{B} \in \mathbb{R}^{K \times N}$. To reduce the number of merge operations, SpArch (Zhang et al. 2020) presents matrix condensing along with the Huffman tree scheduler. First, the matrix condensing is used to reduce the number of partial product maps (Fig. 15a). As nonzeroes are shifted to the leftmost column, elements with the same column index are color-coded for visualization purpose. One drawback with the matrix condensing is that the data reuse factor may reduce at matrix **B** as multiple rows may be needed. To mitigate the fetching overhead, a row prefetcher is utilized that loads the required rows in matrix **B** while streaming in the data from matrix **A**. Even with the matrix condensing a large number of partial product maps can be produced. The optimized merge order significantly reduces the data to be loaded from DRAM, and this is done by the Huffman tree scheduler in SpArch (Fig. 15b). With all these design techniques together, SpArch reduces DRAM access by 2.8× over OuterSPACE.

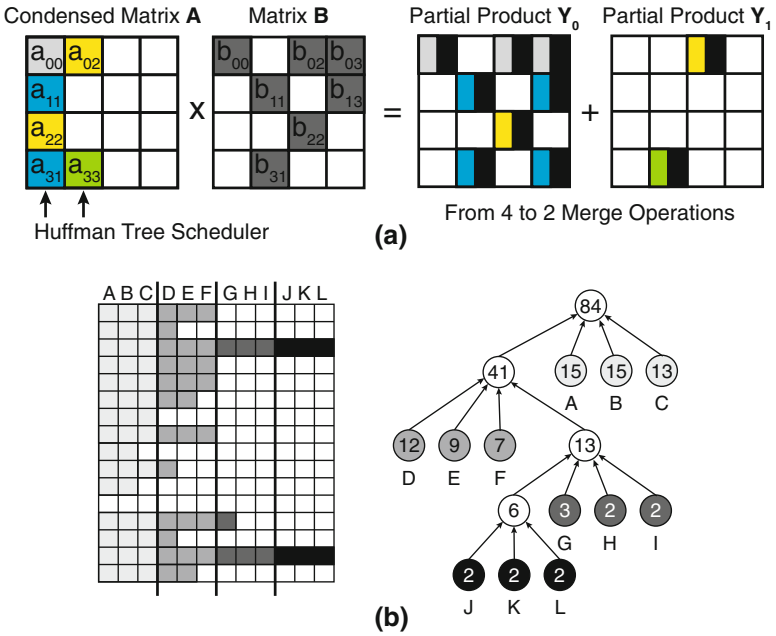


Fig. 15 The hardware accelerator for SpGEMM, named SpArch (Zhang et al. 2020), using matrix condensing, row prefetcher, and the Huffman tree scheduler. (**a**) Condensed outer product operation on the same example from Fig. 13a. (**b**) An example of a four-way Huffman tree scheduler that minimizes the memory traffic

Architectures for Power-Gating-Based Active Leakage Control

In most applications, IoT devices are seldom active. Consider smart home IoT devices such as Google Home (Google Home) or Amazon Blink (Amazon Blink). These devices are rarely fully functional depending on user activity and environmental conditions. In such IoT devices, the overall system will comprise a smaller *always-on* component which continuously listens to user input or environmental activity. However, the majority of the system components need not be constantly active. Nonetheless, if an idle system component is connected to the power grid, even though it doesn't consume any active (i.e., dynamic) power, it will still consume leakage power. Leakage power is dissipated due to various leakage mechanisms in transistors, such as subthreshold leakage, gate tunneling, and body junction leakage. Due to these leakage currents, transistors continue to dissipate static power even when they have been turned off. Especially for the advanced CMOS technologies, leakage power's dissipation can become quite significant due to aggravating short-channel effects in transistors.

Overview of Power-Gating

The leakage power of a system has become a critical concern. To mitigate leakage power, gate control over the channel can be improved using silicon on insulator, FinFET, and nanowire style in transistors. Alternate switching mechanisms such as Tunnel FET (Trivedi and Mukhopadhyay 2014; Trivedi et al. 2014a, 2015) and magnetic (Nasrin et al. 2019) have been explored to operate transistors at lower supply voltage (VDD), thereby also resulting in a lower leakage current. However, in addition to the above technology-level solutions, leakage power can also be minimized by architecture-level techniques and mainly by disconnecting or *power-gating* idle system components from the global power grid (Jiang et al. 2005). Figure 16a shows a p-type power-gating scheme where a top PMOS transistor can disconnect the idle system components from the main power grid to minimize leakage power dissipation. Similarly, an n-type power-gating in Fig. 16b will disconnect idle system components from the ground grid, essentially achieving the same leakage power-saving benefit. Various micro-architectural control signals such as block access signal for caches, clock gating signals for cores, or input/output data phases of LUTs for FPGA can be used to detect if a system component is idle and then apply power-gating to the unit (Hu et al. 2004).

Power-gating can also be performed at various scales. In a coarse-grained power-gating configuration, the power-gating transistor can be shared for all or many system components and can be controlled by a single power-gating signal. On the other hand, in a fine-grained power-gating configuration, the entire system can be partitioned into many power-gating domains, each controlled independently by respective power-gating signals. Compared to coarse-grained power-gating, fine-grained power-gating is more attractive for IoT devices to save leakage energy

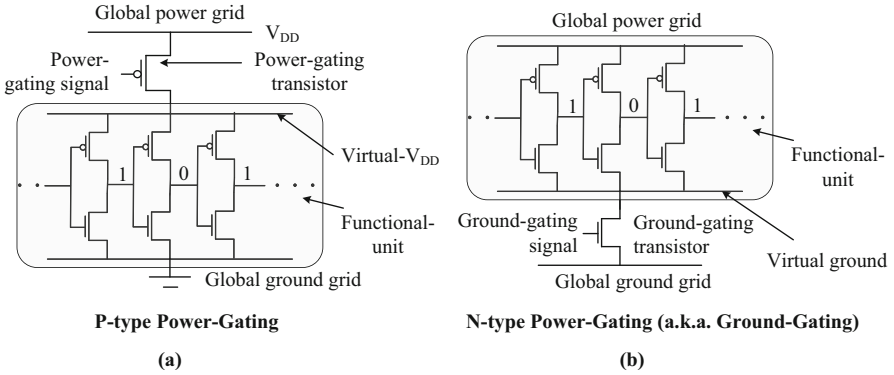


Fig. 16 Power-gating in (**a**) n-type and (**b**) p-type modes to minimize leakage power in a digital system by disconnecting idle components from power/ground grid

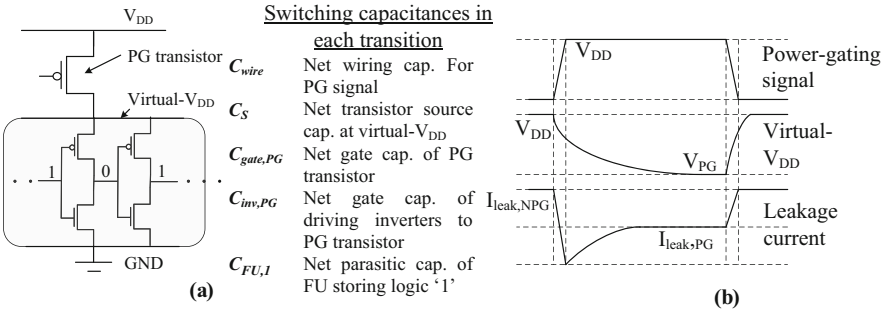


Fig. 17 (**a**) Charging/discharging of additional capacitance due to power-gating configuration. (**b**) Typical profile of leakage current and virtual-VDD due to power-gating

even during brief idle periods. By *finely* detecting the activity of respective system components, fine-grained power-gating schemes can find more opportunities to save leakage power than a coarse-grained scheme that waits for the entire system to be inactive before invoking power-gating.

Challenges and Trade-Offs in Power-Gating

Although power-gating is a simple and effective method to save leakage power, the method comes with a unique set of challenges and trade-offs. Consider power-gating of an inverter chain in Fig. 17a. As the circuit transits between typical and power-gated modes, various additional capacitances, shown to the figure's right, switch, resulting in power-gating energy overheads. For example, the capacitance at virtual-VDD contributes to power-gating overhead. Note that a capacitance at virtual-VDD arises only due to power-gating implementation. In a typical implementation, the

virtual-VDD node will not be present. Furthermore, since many transistors are connected to the virtual-VDD, wire interconnect capacitance, C_{wire}, and transistor parasitic capacitance, C_S, at virtual-VDD are significant and therefore can lead to considerable power-gating overhead. Meanwhile, if power-gating overheads exceed leakage saving, the scheme becomes inefficient.

Figure 17b shows the typical transients of virtual-VDD and leakage current as power-gating is invoked. Virtual-VDD is the local supply node of the power-gated domain which is separated from the global supply grid due to power-gating transistor. In the figure, as soon as the power-gating transistor is turned off, virtual-VDD begins to drop. The leakage current of the logic block at virtual-VDD discharges the node, while the turned-off power-gating transistor does not replenish it at the same rate. Eventually, virtual-VDD settles to a lower voltage V_{PG} where the leakage current of power-gating transistor and underlying logic unit is balanced. Also note that as soon as power-gating is invoked, leakage current from the system drops but then slowly rises to settle to a new level, $I_{leak,PG}$. The power-gating transistor governs the leakage current of a power-gated system. In the beginning, the power-gating transistor is turned off and has a minimal source-to-drain voltage. Therefore, the current through the power-gated unit is relatively small. However, as virtual-VDD settles to V_{PG}, a sufficient source-to-drain voltage develops across the power-gating transistor, resulting in the equilibrium leakage current, $I_{leak,PG}$, through the system.

Voltage variations at the virtual-VDD also induce power overhead at many other circuit nodes. For example, consider the intermediate nodes in the circuit below holding a logic one. As virtual-VDD drops to V_{PG}, these nodes are also discharged. When the circuit becomes active again, all these nodes' potential must be recharged to supply voltage, VDD. Therefore, these logic capacitances, $C_{FU,1}$, also contribute to dynamic energy overheads due to power-gating.

A critical design requirement for the power-gating transistor is also that it should present a minimal resistance when the underlying logic unit is active. If the power-gating transistor's resistance is high, it will reduce the voltage swing (i.e., effective supply voltage) across the logic unit, resulting in a lower performance. Therefore, it is a standard practice to dedicate sufficient area for power-gating transistors to minimize theirs on resistance. Typically, the power-gating of a system leads to ∼5–10% of area overhead. Meanwhile, since the power-gating transistors are large, toggling their gate potential itself becomes energy expensive. Moreover, to switch power-gating transistors with high performance, a dedicated inverter buffer chain is necessary. Both high gate capacitance of power-gating transistor and its driver circuits lead to power-gating overheads as captured by $C_{gate,PG}$ and $C_{inv,PG}$ capacitance in Fig. 17a.

Under a suboptimal design, power-gating can deteriorate the voltage swing at the connected logic unit and induce significant noise at the power grid (Kim et al. 2003). Figure 18 shows this pictorially whereas Logic-1 wakes up, a sudden demand for charging virtual-VDD and intermediate circuit nodes induces a current rush leading to supply voltage droop at the unit. Without a robust power management circuit, voltage droop can also create supply noise spreading throughout the power grid.

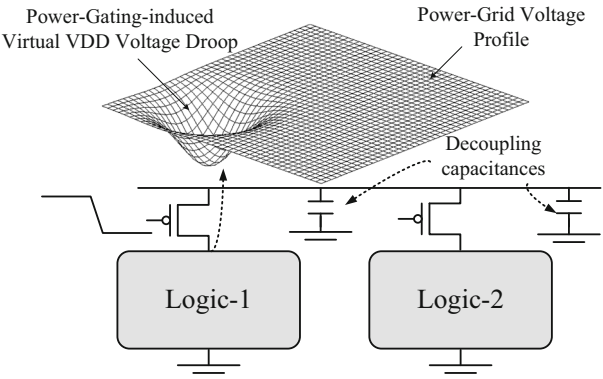


Fig. 18 Power-gating-induced voltage droop and supply noise

To mitigate such power supply noise, various techniques have been proposed. For example, Agarwal et al. (2006) uses multiple sleep modes to adapt against leakage saving and wake-up delay trade-off. Kahng et al. (2013) and Akl et al. (2009) use a *staggered turn on* of power-gating transistors. In the scheme, a power-gating transistor is implemented through an array of parallel instances. When the logic unit activates, the instances of a power-gating transistor are turned on sequentially in a staggered fashion, not all of a sudden, to reduce the current rush. However, due to a staggering turn-on, the power-gated logic unit requires more time to turn on, resulting in performance degradation. In another set of techniques, a decoupling capacitor is added to VDD or virtual-VDD node to minimize the proliferation of supply noise through the grid (Charania et al. 2012). Decoupling capacitances at the power grid reduces the supply grid's time constant, filtering out high-frequency noise. However, this solution also incurs several limitations. First, the placement of sufficient decoupling capacitance incurs a large area. Second, if the decoupling capacitance is placed at the global supply grid, the capacitors are exposed to voltage drop and therefore introduce their own leakage. Alternatively, if the decoupling capacitance is placed at the virtual-VDD, it slows down the power-gating domain's transition between active and inactive modes. Decoupling capacitance at the virtual-VDD also contributes to power-gating overheads.

The leakage energy-saving due to power-gating is strongly dependent on process, voltage, and temperature (PVT) conditions and activity pattern of the logic unit. For example, at high temperatures or low threshold voltage, the leakage current through transistors is high. Therefore, potentially higher leakage power-savings can be achieved under these conditions. Similarly, if transistors are low threshold voltage conditions due to process variability, they will dissipate more leakage, and therefore power-gating can become more effective. When the overheads of power-gating exceed the leakage energy-saving, power-gating becomes inefficient. A power-gating scheme that doesn't intrinsically account for such leakage power-saving and transition energy overhead trade-offs is bound to be suboptimal under varying

PVT and system activity conditions. Even more, if the power-gating architecture is excessively complex, it will have its significant energy overheads, reducing the benefit of power-gating.

Overall, even though power-gating is an elegant and straightforward approach to minimize leakage power wastage in IoT devices, the technique requires several deeper considerations, as illustrated above. In the following, a unique *learning-based* approach is discussed to dynamically characterize trade-offs between energy-saving and power-gating overhead so that a self-adaptive architecture only invokes power-gating when it is efficient to do so.

Power-Gating Efficiency Learner

Since the efficiency of power-gating strongly depends on PVT conditions of the system, which can vary dynamically, as well as input activity patterns, an intelligent power-gating architecture must continuously adapt to these variations. In the following, the design of a power-gating efficiency learner is discussed to characterize benefits of power-gating under such dynamic trade-offs. Consider the learner design as shown in Fig. 19a for a p-type power-gated system (Trivedi et al. 2014b; Trivedi and Mukhopadhyay 2012). The learner is controlled by the idle

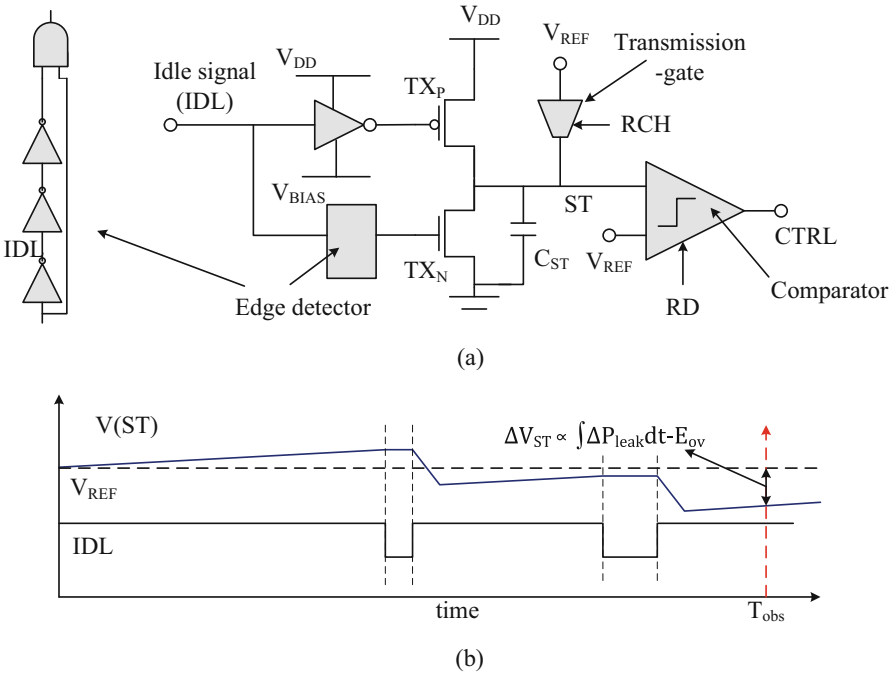


Fig. 19 (**a**) Power-gating efficiency learner circuit. (**b**) Learning transients

signal of the power-gated domain, i.e., IDL. When IDL signal transitions from zero to one (0→1), it indicates a power-gating opportunity for the system. Mode transitions in a power-gated domain incurs energy overhead ($E_{ov,tran}$); meanwhile, leakage savings depend on PVT conditions as well as the duration/activity of IDL signal. Power-gating efficiency learner mimics this behavior using a two-transistor replica of the power-gated domain. With a 0→1 transition in IDL signal, transistor TX_N in the learner circuit is activated through the edge detector and reduces the voltage (V_{ST}) of the node ST. On the other hand, when IDL = 1, the subthreshold biased transistor TX_P increases V_{ST}. Therefore, TX_N and TX_P contend to regulate the potential of the node ST. The transition energy of power-gated domain, i.e., $E_{ov/tran}$, is reflected by the drop in V_{ST} induced by TX_N, whereas the leakage energy-saving $\int \Delta P_{leak} dt$ is reflected by the increase in V_{ST} induced by TX_P. TX_N and TX_P can be designed to ensure that V_{ST} follows $E_{ov/tran}$ and ΔP_{leak} in the power-gated domain. This can be achieved by designing them based on the following equations:

$$I(TX_P) \propto \Delta P_{leak} \tag{3a}$$

$$T_{PW} \times I(TX_N) \propto E_{ov/tran} \tag{3b}$$

$$\frac{I(TX_{NP})}{T_{PW} \times I(TX_N)} = \frac{\Delta P_{leak}}{E_{ov/tran}} = \frac{V_{DD} \times (I_{NPG} - I_{PG})}{E_{ov/tran}} \tag{3c}$$

Here, $I(TX_P)$ is the current through TX_P. T_{PW} is the pulse width generated by the edge detector. $I(TX_N)$ is the on-current of TX_N. I_{NPG} and I_{PG} are the leakage currents of the power-gated domain in the non-power-gated and power-gated modes, respectively. To minimize the area of the learner circuit, TX_N can be of the minimum size, and TX_P can be determined by Eq. 3c. By following the above equations, the learner circuit also intrinsically tracks the PVT variations in power domain in arbitrating power-gating efficiency. Due to its small area, the learner circuit can be embedded within the power domain to sense local PVT conditions. If the power-gated domain temperature rises, the current through subthreshold biased TX_P also increases, resulting in a faster charging rate of the node ST. Therefore, under the same activity pattern of TX_N, node ST's potential rises faster at higher temperatures. Similarly, if the power-gated domain is explored to low threshold voltage, TX_P in the embedded learner circuit will also have a lower threshold voltage. Thus, with varying process conditions, the charging rate from TX_P tracks the process corner.

Self-Adaptive Power-Gating Architecture

Figure 20 shows a self-adaptive power-gating architecture that utilizes power-gating efficiency learner's output to minimize leakage energy while being considerate of

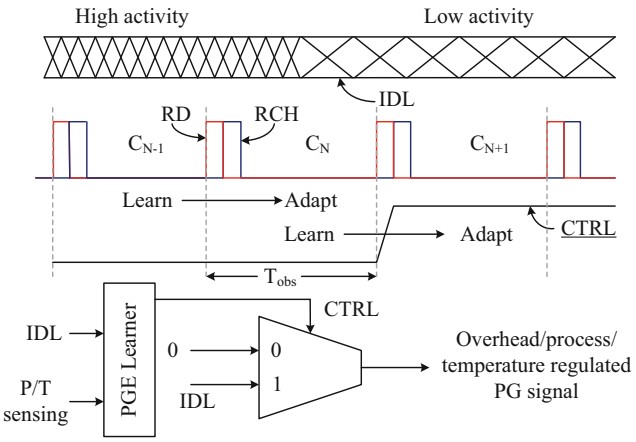


Fig. 20 Self-adaptive power-gating based on power-gating efficiency learner circuit

its overheads. In the figure, the periodic signal RD determines the characterization cycle (C_N) of the learner. In this period, by comparing potential leakage savings to power-gating overheads, the learner deduces if it is beneficial to perform power-gating. At the beginning of C_N, the node ST is charged to the reference voltage (V_{REF}) through the transmission gate in the learner circuit and using the control signal RCH. At the end of C_N, the final ST voltage V_{ST} is compared to V_{REF} using a clocked comparator-controlled by the read signal RD. Based on such a comparison, the learner generates the output signal CTRL. If CTRL = 0, i.e., $V_{REF} > V_{ST}$, it indicates that the overheads are dominant since TX_N is able to discharge C_{ST} faster than TX_P can charge it. Essentially, overheads in the power-gated domain are dominant over the energy-savings under power-gating. In this case, the multiplexer in Fig. 20 does not perform power-gating even if the unit is idle. On the other hand, if CTRL = 1, it indicates that leakage savings are dominant over the overheads. Therefore, power-gating is performed as usual depending on the idle signal. Since learner circuit's output intrinsically depends on PVT conditions as well as activity patterns, power-gating is adaptive to such variations in the power-gated domain.

Test Chip and Measurement Results

Figure 21a shows the characterization of power-gating efficiency learner circuit that was demonstrated in our prior work using a test chip designed in 130-nm CMOS (Trivedi et al. 2014b). An internal signal generator generates the read and recharge (RD/RCH) signals for the learner in the chip. The IDL signals can be generated internally or supplied externally through FPGA. Four power-gating domains are implemented with regular-V_{TH} (RVT) and low-V_{TH} (LVT) transistors. Each domain consisted of a chain of 301 inverters to emulate a very fine-grained

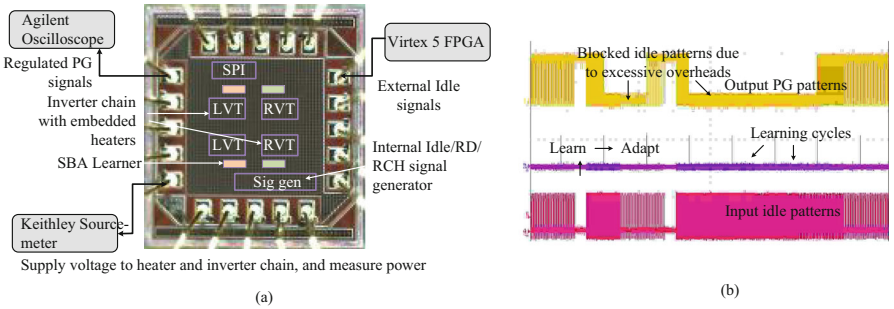


Fig. 21 (**a**) Test chip to characterize power-gating efficiency learner. (**b**) Self-adaptation under varying idle signal patterns

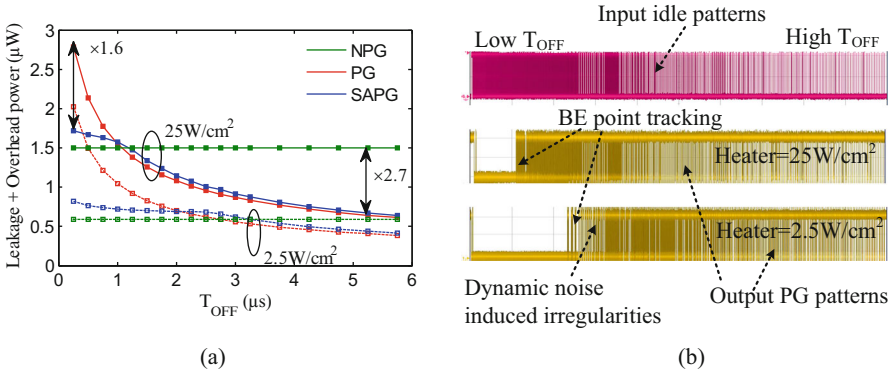


Fig. 22 (**a**) Leakage + overhead power at various power-gating scenarios (*NPG* no power-gating, *PG* power-gating, *SAPG* self-adaptive power-gating). (**b**) Measured generation of idle signal-based power-gating signal at varying idle signal activity and temperature

power-gating domain. The RVT and LVT power-gating domains essentially emulate extreme within-chip process variations. A heater, designed with diffusion resistors, is embedded in the design to emulate dynamic temperature variations. Heater power is varied to control the on-chip temperature, essentially emulating the effect of hot spots and local temperature variations.

A key metric for power-gating-induced energy benefit is *break-even* time. After the onset of power-gating, at break-even point, leakage energy-saving equalizes to power-gating overhead. If the continuous idle period is longer than the break-even point, power-gating is rewarding. The learner circuit's accuracy is characterized by comparing the actual break-even point of a power-gated domain against the one predicted by the learner. The actual breakeven is measured by directly power-gating the domain with the periodic idle patterns (IDL) of fixed on time and varying off time (T_{OFF}). At a lower T_{OFF}, the average total (leakage + overhead) energy of the domain increases. The corresponding results (label: PG) are shown in Fig. 22a. The non-power-gated (label: NPG) case is the leakage power in the absence of

power-gating. The measured (or actual) breakeven of the power-gated domain is defined as the T_{OFF} at which the NPG and PG curves intersect. For an idle pattern with T_{OFF} = breakeven, the overhead is equal to the leakage saving. The measured breakeven for the LVT design is ∼1 and ∼2.7 µs for 25 and 2.5 W/cm^2 heater powers, respectively.

The breakeven predicted by the learner is measured by applying a similar IDL signal and observing the output power-gating signal. The measured waveforms in Fig. 22b show that the learner blocks filter out IDL patterns with very low T_{OFF}. The IDL patterns with higher T_{OFF} pass through the learner. The learner's predicted breakeven is defined as the T_{OFF} of IDL when CTRL makes the 0→1 transition. The heater power (or temperature)-dependent BE tracking is also shown in the figure. At the lower heater power (or lower temperature), the learner correctly identifies the breakeven at a higher T_{OFF} value.

The discussed self-adaptive power-gating scheme can dynamically adapt to varying activity patterns. Nonetheless, the learning efficacy is strongly dependent on its learning cycle (i.e., the period of RD pulses). If the learning period is large, the self-adaptive scheme cannot swiftly adapt to sudden changes in domain's activity patterns. Consider an example case in Fig. 23a where power-gating opportunity at low activity is missed since the scheme does not quickly adapt to changes in the activity pattern. Meanwhile, if the learning period is minimal, the self-adaptive procedure can pose considerable overhead, thereby eclipsing power-gating benefits. Figure 23b illustrates this based on power measurements at varying learning period. In the figure, two test cases of IDL signal activity pattern are considered: a periodic IDL signal and another based on a pseudorandom sequence (PRS). Note that for PRS idle pattern, both minimal and extensive learning periods are suboptimal due to the reasons above. Fortunately, the self-adaptive scheme's unique advantage is that its operation does not depend on the learning cycle period. The learning cycle can be controlled on demand using top-level controllers. Top-level controllers can analyze the statistics of various applications on IoT devices and derive a corresponding optimal learning period for the self-adaptive scheme.

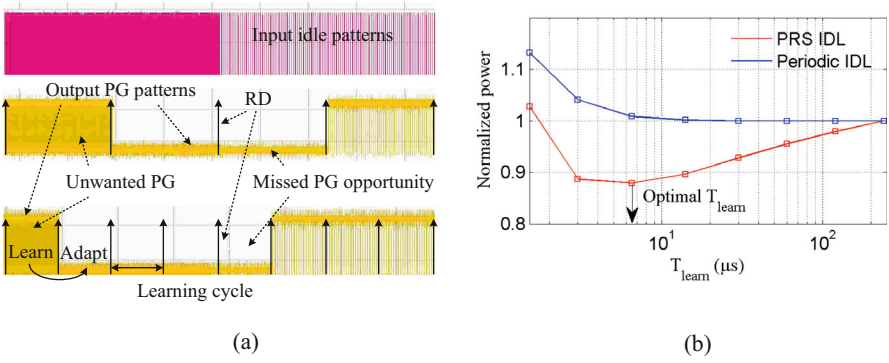


Fig. 23 (**a**) The effect of learning cycle on self-adaptive power-gating scheme. (**b**) Overall system energy minimizes at an optimal learning period

Conclusion and Future Roadmap

This chapter has reviewed complementary techniques toward pervasive self-powered edge intelligence. The *first design principle* discussed in this chapter was to exploit integrated sensors in edge devices to scavenge energy from the application environment, thereby enhancing on-edge energy resources. The test case of CMOS sensor array that opportunistically harvests energy when the edge node is not operating is discussed. An AMM unit was presented for this purpose. The impact of factors such as processing pipeline, unit pixel size, and leakage energy on self-powered operations was also considered. The discussed design was able to harvest $\sim$2.15 μW power from the environment. Future designs can substantially enhance the harvesting efficiency by nontraditional sensor materials and processes that are optimized for both signal acquisition and harvesting (Dong et al. 2020a; Zohair et al. 2020; Dong et al. 2020b; Shehata et al. 2020). Likewise, more optimal power converter designs, such as those integrating both boost and buck stages and adaptively switching their operation mode (Hu et al. 2020; Narasimman et al. 2020; Singh and Fayed 2020), can improve the efficiency of power management under harvesting.

The *second design principle* discussed in this chapter was to minimize the computational demand itself. Specifically, resource-aware image sensor architectures based on ROI-based coding and rate control for better energy-quality scalability are reviewed. In addition to the design of energy-efficient sensor-based systems, the benefit of exploiting sparsity is discussed in both inputs and parametric models when realizing the edge intelligence. To achieve the maximum utilization of processing units, many architectural innovations were made to efficiently process the sparse data. As the computing models for IoT devices are often sparse, placing sparse processors in edge devices will significantly improve the battery lifetime. Future designs can further improve the computational efficiency based on this design principle by integrating end-to-end learning frameworks on ROI identification, such as attention mechanisms in deep neural networks (Veličković et al. 2017; Gulcehre et al. 2018; Liu et al. 2019). Similarly, model sparsity can be further exploited by operating in compressed parameter domain itself (Han et al. 2016c; Parashar et al. 2017; Zhang et al. 2016) or by adapting inference operators such as by operating on binary or low-precision weights (Lin et al. 2017; Nasrin et al. 2020, 2021a,b).

Finally, the *third design principle* discussed in this chapter was to minimize the wastage of energy resources. Since edge devices typically have a low activity, therefore, they are susceptible to energy wastage due to various leakage mechanisms. Learning-based architectures for active leakage control are reviewed. The discussed techniques enabled self-adaptation against various dynamic factors such as temperature, process, and voltage variation as well as activity patterns of the edge device. The future techniques can further enhance leakage energy-savings by even more fine-grained and lightweight power-gating architectures (Boyapati et al. 2017), by synergistically integrating power-gating with power regulation (Uzun and Köse 2014), and by lightweight time-domain signal analytics (Shylendra et al. 2020a,b) to better adapt against even higher order activity and operating condition variations.

References

Agarwal K, Deogun H, Sylvester D, Nowka K (2006) Power gating with multiple sleep modes. In: 7th international symposium on quality electronic design (ISQED'06). IEEE, p 5

Akl CJ, Ayoubi RA, Bayoumi MA (2009) An effective staggered-phase damping technique for suppressing power-gating resonance noise during mode transition. In: 2009 10th international symposium on quality electronic design. IEEE, pp 116–119

Amazon Blink. https://www.amazon.com/stores/page/C5DECBBE-4F56-4C36-B933-E62144578691

Anandkumar A, Ge R, Hsu D, Kakade SM, Telgarsky M (2014) Tensor decompositions for learning latent variable models. J Mach Learn Res 15(80):2773–2832. [Online]. Available: http://jmlr.org/papers/v15/anandkumar14b.html

Arora S, Leighton T, Maggs B (1990) On-line algorithms for path selection in a nonblocking network. In: Proceedings of ACM symposium on theory of computing (STOC), pp 149–158

Bank RE, Douglas CC (1993) Sparse matrix multiplication package (SMMP). Adv Comput Math 1:127–137

Benezeth Y, Jodoin P-M, Emile B, Laurent H, Rosenberger C (2010) Comparative study of background subtraction algorithms. J Electron Imag 19(3):033003

Bennett J, Lanning S (2007) The Netflix prize. In: KDD cup and workshop in conjunction with KDD

Boyapati R, Huang J, Wang N, Kim KH, Yum KH, Kim EJ (2017) Fly-over: a light-weight distributed power-gating mechanism for energy-efficient networks-on-chip. In: 2017 IEEE international parallel and distributed processing symposium (IPDPS). IEEE, pp 708–717

Brutzer S, Höferlin B, Heidemann G (2011) Evaluation of background subtraction techniques for video surveillance. In: IEEE CVPR, pp 1937–1944. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5995508

Cevik I, Huang X, Yu H, Yan M, Ay S (2015) An ultra-low power CMOS image sensor with on-chip energy harvesting and power management capability. Sensors 15(3):5531–5554. [Online]. Available: http://www.mdpi.com/1424-8220/15/3/5531/

Charania T, Opal A, Sachdev M (2012) Analysis and design of on-chip decoupling capacitors. IEEE Trans Very Large Scale Integr (VLSI) syst 21(4):648–658

Chefi A, Soudani A, Sicard G (2013) A CMOS image sensor with low-complexity video compression for wireless sensor networks. In: IEEE NEWCAS, pp 1–4. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6573591 http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6573591

Chiou AY-C, Hsieh C-C (2015) A 0.4 V self-powered CMOS imager with 140 dB dynamic range and energy harvesting C86 C87, pp 86–87

Dong L, Jin C, Closson AB, Trase I, Richards HR, Chen Z, Zhang JX (2020a) Cardiac energy harvesting and sensing based on piezoelectric and triboelectric designs. Nano Energy 76:105076

Dong L, Closson AB, Jin C, Nie Y, Cabe A, Escobedo D, Huang S, Trase I, Xu Z, Chen Z et al (2020b) Multifunctional pacemaker lead for cardiac energy harvesting and pressure sensing. Adv Healthcare Mater 9(11):2000053

Dunne MC, Potts RB (1964) Algorithm for traffic control. Oper Res 12(6):870–881

Eisenstat SC, Elman HC, Schultz MH, Sherman AH (1984) The (new) Yale sparse matrix package. In: Elliptic Problem Solvers, vol 2, pp 45–52

Elgammal A, Harwood D, Davis L (2000) Non-parametric model for background subtraction. In: ECCV, vol 1843, pp 751–767

Fallah YP, Mansour H, Khan S (2008) A link adaptation scheme for efficient transmission. Circuits Syst Video Technol IEEE 18(7):875–887

Gao W, Hsu D, Lee WS, Shen S, Subramanian K (2017) Intention-Net: integrating planning and deep learning for goal-directed autonomous navigation, CoRR, vol. abs/1710.05627. [Online]. Available: http://arxiv.org/abs/1710.05627

Google Home. https://store.google.com/us/magazine/compare_nest_speakers_displays

Grasedyck L, Kressner D, Tobler C (2013) A literature survey of low-rank tensor approximation techniques

Grois D, Hadar O (2014) Complexity-aware adaptive preprocessing scheme for region-of-interest spatial scalable video coding. IEEE Trans Circuits Syst Video Technol 24(6):1025–1039. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6727577

Grois D, Member S, Hadar O (2011) Efficient adaptive bit-rate control for scalable video coding by using computational complexity-rate-distortion analysis. In: The IEEE international symposium on broadband multimedia systems and broadcasting (BMSB)

Gulcehre C, Denil M, Malinowski M, Razavi A, Pascanu R, Hermann KM, Battaglia P, Bapst V, Raposo D, Santoro A et al (2018) Hyperbolic attention networks, arXiv preprint arXiv:1805.09786

Hampapur A, Brown L, Connell J, Pankanti S, Senior A, Tian Y (2003) Smart surveillance applications, technologies and implications.pdf. In: ICICS-FCM

Han S, Mao H, Dally WJ (2016a) Deep compression: compressing deep neural network with pruning, trained quantization and Huffman coding. In: International conference on learning representations (ICLR). [Online]. Available: http://arxiv.org/abs/1510.00149

Han S, Liu X, Mao H, Pu J, Pedram A, Horowitz MA, Dally WJ (2016b) EIE: efficient inference engine on compressed deep neural network. In: Proceedings of international symposium on computer architecture (ISCA), pp 243–254

Han S, Liu X, Mao H, Pu J, Pedram A, Horowitz MA, Dally WJ (2016c) Eie: efficient inference engine on compressed deep neural network. ACM SIGARCH Comput Archit News 44(3):243–254

Hapla V, Horak D, Merta M (2013) Use of direct solvers in TFETI massively parallel implementation. Springer, Berlin/Heidelberg, pp 192–205

Haratcherev L, Taal J (2005) Fast 802.11 link adaptation for real-time video streaming by cross-layer signaling. In: IEEE international symposium on circuits and systems. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1465389

Hegde K, Asghari-Moghaddam H, Pellauer M, Crago N, Jaleel A, Solomonik E, Emer J, Fletcher CW (2019) ExTensor: an accelerator for sparse tensor algebra. In: Proceedings of IEEE/ACM international symposium on microarchitecture (MICRO), pp 319–333

Horowitz M. Energy table for 45 nm process, Stanford VLSI Wiki. [Online]. Available: https://sites.google.com/site/seecproject

Hu Z, Buyuktosunoglu A, Srinivasan V, Zyuban V, Jacobson H, Bose P (2004) Microarchitectural techniques for power gating of execution units. In: Proceedings of the 2004 international symposium on low power electronics and design, pp 32–37

Hu Y, Meng F, Wang Y (2012) Improved JPEG compression algorithm based on saliency maps. In: CISP

Hu K-Y, Tsai C-H, Tsai C-W (2020) Digital v2 constant on-time control buck converter with adaptive voltage positioning and automatic calibration mechanism. IEEE Trans Power Electron 36: 7178–7188

Huang C, Lin C (2009) Multiple-priority region-of-interest H.264 video compression using constraint variable bitrate control for video surveillance. Opt Eng 48(4):047004. [Online]. Available: https://doi.org/10.1117/1.3120485

Imran M, Ahmad N, Khursheed K, Waheed MA, Lawal N, O'Nils M (2013) Implementation of wireless vision sensor node with a lightweight bi-level video coding. IEEE JETCAS 3(2):198–209

Jiang H, Marek-Sadowska M, Nassif SR (2005) Benefits and costs of power-gating technique. In: 2005 international conference on computer design. IEEE, pp 559–566

Jouppi NP, Young C, Patil N, Patterson D, Agrawal G, Bajwa R, Bates S, Bhatia S, Boden N, Borchers A, Boyle R, Cantin P, Chao C, Clark C, Coriell J, Daley M, Dau M, Dean J, Gelb B, Ghaemmaghami TV, Gottipati R, Gulland W, Hagmann R, Ho CR, Hogberg D, Hu J, Hundt R, Hurt D, Ibarz J, Jaffey A, Jaworski A, Kaplan A, Khaitan H, Killebrew D, Koch A, Kumar N, Lacy S, Laudon J, Law J, Le D, Leary C, Liu Z, Lucke K, Lundin A, MacKean G, Maggiore A, Mahony M, Miller K, Nagarajan R, Narayanaswami R, Ni R, Nix K, Norrie T, Omernick M,

Penukonda N, Phelps A, Ross J, Ross M, Salek A, Samadiani E, Severn C, Sizikov G, Snelham M, Souter J, Steinberg D, Swing A, Tan M, Thorson G, Tian B, Toma H, Tuttle E, Vasudevan V, Walter R, Wang W, Wilcox E, Yoon DH (2017) In-datacenter performance analysis of a tensor processing unit. In: ACM/IEEE international symposium on computer architecture (ISCA), pp 1–12

Kahng AB, Kang S, Rosing TS, Strong R (2013) Many-core token-based adaptive power gating. IEEE Trans Comput-Aided Des Integr Circuits Syst 32(8):1288–1292

Kim C, Hwang J-N (2002) Fast and automatic video object segmentation and tracking for content-based applications. IEEE TCSVT 12(2):122–129. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=988659

Kim S, Kosonocky SV, Knebel DR (2003) Understanding and minimizing ground bounce during mode transition of power gating structures. In: Proceedings of the 2003 international symposium on low power electronics and design, pp 22–25

Kim K, Chalidabhongse TH, Harwood D, Davis L (2005) Real-time foreground-background segmentation using codebook model. Real-Time Imag 11(3):172–185. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S1077201405000057

Kim G, Lee Y, Foo Z, Pannuto P, Kuo YS, Kempke B, Ghaed MH, Bang S, Lee I, Kim Y, Jeong S, Dutta P, Sylvester D, Blaauw D (2014) A millimeter-scale wireless imaging system with continuous motion detection and energy harvesting. In: IEEE symposium on VLSI circuits, digest of technical papers, no. Dec 2011, pp 31–32

Ko JH, Mukhopadhyay S (2016) An energy-aware approach to noise-robust moving object detection for low-power wireless image sensor platforms. In: International symposium on low power electronics and design (ISLPED)

Ko JH, Ahmed KZ, Amir MF, Na T, Mukhopadhyay S (2017) A single-chip image sensor node with energy harvesting from CMOS pixel array. IEEE Trans Circuits Syst I, Reg Papers64(9):2295–2307

Ko JH, Mudassar BA, Mukhopadhyay S (2015) An energy-efficient wireless video sensor node for moving object surveillance. IEEE TMSCS 1(1):7–18

Ko JH, Na T, Mukhopadhyay S (2016) An energy-efficient wireless video sensor node with a region-of-interest based multi-parameter rate controller for moving object surveillance. In: IEEE advanced video and signal-based surveillance (AVSS), pp 138–144

Kung J, Park J, Park S, Kim J-J (2019) Peregrine: a flexible hardware accelerator for LSTM with limited synaptic connection patterns. In: Proceedings of the 56th annual design automation conference (DAC)

Lai W, Gu X-D, Wang R-H, Dai L-R, Zhang H-J (2004) A region based multiple frame-rate tradeoff of video streaming. In: 2004 international conference on image processing, 2004. ICIP'04, vol 3, pp 2067–2070. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1421491

Law MK, Bermak A, Shi C (2011) A low-power energy-harvesting logarithmic CMOS image sensor with reconfigurable resolution using two-level quantization scheme. IEEE Trans Circuits Syst II: Express Briefs 58(2):80–84

Lin X, Zhao C, Pan W (2017) Towards accurate binary convolutional neural network, arXiv preprint arXiv:1711.11294

Liu T, Qi Y, Shi L, Yan J (2019) Locate-then-detect: real-time web attack detection via attention-based deep neural networks. In: IJCAI, pp 4725–4731

Mattson T, Bader D, Berry J, Buluc A, Dongarra J, Faloutsos C, Feo J, Gilbert J, Gonzalez J, Hendrickson B, Kepner J, Leiserson C, Lumsdaine A, Padua D, Poole S, Reinhardt S, Stonebraker M, Wallach S, Yoo A (2013) Standards for graph algorithm primitives. In: IEEE high performance extreme computing conference (HPEC), pp 1–2

McKenna SJ, Raja Y, Gong S (1999) Tracking colour objects using adaptive mixture models. Image Vis Comput 17(3–4):225–231. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0262885698001048

Meddeb M, Cagnazzo M, Pesquet-Popesc B (2014) Region-of-interest based rate control scheme for high efficiency video coding. In: APSIPA transactions on signal and information processing, pp 1–9

Narasimman N, Salahuddin R, Singh RP (2020) An 86% efficiency multi-phase buck converter using time-domain compensator and adaptive dead-time control for DVS application. In: IECON 2020 the 46th annual conference of the IEEE industrial electronics society. IEEE, pp 2255–2260

Nasrin S, Drobitch JL, Bandyopadhyay S, Trivedi AR (2019) Low power restricted boltzmann machine using mixed-mode magneto-tunneling junctions. IEEE Electron Device Lett 40(2):345–348

Nasrin S, Ramakrishna S, Tulabandhula T, Trivedi AR (2020) Supported-binarynet: bitcell array-based weight supports for dynamic accuracy-energy trade-offs in sram-based binarized neural network. In: 2020 IEEE international symposium on circuits and systems (ISCAS). IEEE, pp 1–5

Nasrin S, Badawi D, Cetin A, Gomes W, Trivedi AR (2021a) MF-Net: compute-in-memory sram for multibit precision inference using memory-immersed data conversion and multiplication-free operators. IEEE Trans Circuits Syst-I 68:1966–1978

Nasrin S, Shukla P, Jaisimha S, Trivedi AR (2021b) Compute-in-memory upside down: a learning operator co-design perspective for scalability. In: IEEE design automation and test in Europe (DATE)

Naumov M, Mudigere D, Shi HM, Huang J, Sundaraman N, Park J, Wang X, Gupta U, Wu C, Azzolini AG, Dzhulgakov D, Mallevich A, Cherniavskii I, Lu Y, Krishnamoorthi R, Yu A, Kondratenko V, Pereira S, Chen X, Chen W, Rao V, Jia B, Xiong L, Smelyanskiy M (2019) Deep learning recommendation model for personalization and recommendation systems, CoRR, vol. abs/1906.00091. [Online]. Available: http://arxiv.org/abs/1906.00091

Nayar SK, Sims DC, Fridberg M (2015) Towards self-powered cameras. In: 2015 IEEE international conference on computational photography (ICCP), pp 1–10

Oliver NM, Rosario B, Pentland AP (2000) A bayesian computer vision system for modeling human interactions. IEEE Trans Pattern Anal Mach Intell 22(8):831–843

Ordejon P (1998) Order-N tight-binding methods for electronic-structure and molecular dynamics. Comput Mater Sci 12(3):157–191

Pal S, Beaumont J, Park D, Amarnath A, Feng S, Chakrabarti C, Kim H, Blaauw D, Mudge T, Dreslinski R (2018) OuterSPACE: an outer product based sparse matrix multiplication accelerator. In: IEEE international symposium on high performance computer architecture (HPCA), pp 724–736

Parashar A, Rhu M, Mukkara A, Puglielli A, Venkatesan R, Khailany B, Emer J, Keckler SW, Dally WJ (2017) SCNN: an accelerator for compressed-sparse convolutional neural networks. ACM SIGARCH Comput Archit News 45(2):27–40

Qin E, Samajdar A, Kwon H, Nadella V, Srinivasan S, Das D, Kaul B, Krishna T (2020) SIGMA: a sparse and irregular GEMM accelerator with flexible interconnects for DNN training. In: IEEE international symposium on high performance computer architecture (HPCA), pp 58–70

Robinson HA, Cherry C (1967) Results of a prototype television bandwidth compression scheme. Proc IEEE 55(3):356–364

Shehata N, Hassanin AH, Elnabawy E, Nair R, Bhat SA, Kandas I (2020) Acoustic energy harvesting and sensing via electrospun pvdf nanofiber membrane. Sensors 20(11):3111

Shukla P, Muralidhar A, Iliev N, Tulabandhula T, Fuller SB, Trivedi AR (2021) Ultralow-power localization of insect-scale drones: interplay of probabilistic filtering and compute-in-memory. IEEE Trans Very Large Scale Integr (VLSI) Syst 30:68–80

Shylendra A, Shukla P, Mukhopadhyay S, Bhunia S, Trivedi AR (2020a) Low power unsupervised anomaly detection by nonparametric modeling of sensor statistics. IEEE Trans Very Large Scale Integr (VLSI) Syst 28(8):1833–1843

Shylendra A, Alizad SH, Shukla P, Trivedi AR (2020b) Non-parametric statistical density function synthesizer and monte carlo sampler in CMOS. In: 2020 33rd international conference on VLSI design and 2020 19th international conference on embedded systems (VLSID). IEEE, pp 19–24

Singh M, Fayed AA (2020) A 1-a 6-mhz digitally assisted buck–boost converter with seamless mode transitions and fast dynamic performance for mobile devices. IEEE Trans Power Electron 36(4):4338–4351

Trivedi AR, Mukhopadhyay S (2012) Self-adaptive power gating with test circuit for on-line characterization of energy inflection activity. In: 2012 IEEE 30th VLSI test symposium (VTS). IEEE, pp 38–43

Trivedi AR, Mukhopadhyay S (2014) Potential of ultralow-power cellular neural image processing with Si/Ge tunnel FET. IEEE Trans Nanotechnol 13(4):627–629

Trivedi AR, Amir MF, Mukhopadhyay S (2014a) Ultra-low power electronics with si/ge tunnel FET. In: 2014 design, automation & test in Europe conference & exhibition (DATE). IEEE, pp 1–6

Trivedi AR, Yueh W, Mukhopadhyay S (2014b) In situ power gating efficiency learner for fine-grained self-adaptive power gating. IEEE Trans Circuits Syst II: Express Briefs 61(5):344–348

Trivedi A, Pandey R, Liu H, Datta S, Mukhopadhyay S (2015) Gate/source overlapped hetero-junction tunnel FET for non-boolean associative processing with plasticity. In: 2015 IEEE international electron devices meeting (IEDM). IEEE, pp 17–8

Tsapatsoulis N, Loizou C, Pattichis C (2007) Region of interest video coding for low bit-rate transmission of carotid ultrasound videos over 3G wireless networks. In: Annual international conference of the IEEE engineering in medicine and biology, pp 3717–3720

Tuan M-C, Chen S-L (2015) Fully pipelined VLSI architecture of a real-time block-based object detector for intelligent video surveillance systems. In: 2015 IEEE/ACIS 14th international conference on computer and information science (ICIS), pp 149–154. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7166585

Uzun OA, Köse S (2014) Converter-gating: a power efficient and secure on-chip power delivery system. IEEE J Emerg Sel Top Circuits Syst 4(2):169–179

Vasudevan A, Anderson A, Gregg D (2017) Parallel multi channel convolution using general matrix multiplication. In: IEEE international conference on application-specific systems, architectures and processors (ASAP), pp 19–24

Veličković P, Cucurull G, Casanova A, Romero A, Lio P, Bengio Y (2017) Graph attention networks, arXiv preprint arXiv:1710.10903

Wang H-T, Leon-Salas WD (2015) An image sensor with joint sensing and energy harvesting functions. IEEE Sens J 15(2):902–916. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6894563

Zhang S, Du Z, Zhang L, Lan H, Liu S, Li L, Guo Q, Chen T, Chen Y (2016) Cambricon-x: an accelerator for sparse neural networks. In: 2016 49th annual IEEE/ACM international symposium on microarchitecture (MICRO). IEEE, pp 1–12

Zhang Z, Wang H, Han S, Dally WJ (2020) SpArch: efficient architecture for sparse matrix multiplication. In: IEEE international symposium on high performance computer architecture (HPCA), pp 261–274

Zhou Z, Chen X, Li E, Zeng L, Luo K, Zhang J (2019) Edge intelligence: paving the last mile of artificial intelligence with edge computing. Proc IEEE 107(8):1738–1762

Zohair M, Moyer K, Eaves-Rathert J, Meng C, Waugh J, Pint CL (2020) Continuous energy harvesting and motion sensing from flexible electrochemical nanogenerators: toward smart and multifunctional textiles. ACS Nano 14(2):2308–2315

Real-Time Scheduling for Computing Architectures

4

Arvind Easwaran, Michael Yuhas, Saravanan Ramanathan, and Ankita Samaddar

Contents

Real-Time Operating System (RTOS) . . . 128
Introduction to Key OS Features . . . 128
Introduction to Real-Time Systems . . . 130
Real-Time CPU Scheduling . . . 132
Scheduling on Single-Core CPUs . . . 132
Scheduling on Multi-core CPUs . . . 134
Real-Time Scheduling for CPU-GPU Systems . . . 137
GPU Background . . . 137
Scheduling Tasks on a Single GPU . . . 140
Multi-GPU and CPU-GPU Scheduling . . . 143
Application Domains . . . 146
Tools and Frameworks . . . 147
Alternative Architectures . . . 148
Real-Time Edge Computing Systems . . . 148
Introduction to Edge Computing . . . 148
The Edge Architecture . . . 150
Real-Time Edge Computing . . . 151
Resource Allocation in Real-Time Edge . . . 152
Introduction to Real-Time Networks . . . 156
Real-Time Wired Networks . . . 156
Real-Time Wireless Networks . . . 158
Real-Time Flow . . . 161
Routing and Scheduling in Real-Time Wireless Sensor Networks . . . 163
Summary . . . 164
References . . . 165

A. Easwaran (✉) · M. Yuhas · S. Ramanathan · A. Samaddar
Nanyang Technological University, Singapore, Singapore
e-mail: arvinde@ntu.edu.sg; michaelj004@e.ntu.edu.sg; saravana016@e.ntu.edu.sg; ankita003@e.ntu.edu.sg

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_5

Abstract

An operating system (OS) is a supervisory program in a computing system, responsible for efficient management of the hardware resources. In the context of real-time systems, that is, systems in which timeliness and predictability in the worst case are critical, the real-time OS (RTOS) additionally has to ensure satisfaction of all hard deadlines in the system. This chapter considers the RTOS resource scheduling problem in a variety of computing architectures including single-core central processing units (CPUs), multi-core CPUs, CPUs with graphics processing units (GPUs) as co-processors and distributed edge servers. In particular, seminal literature addressing the problem of real-time scheduling of the processing capacity in these architectures is presented. A review of important resource management techniques for wireless and wired networks with real-time requirements is also presented, since such networks are essential for the predictable transmission of workload in the distributed edge.

Keywords

Real-time operating system · Task scheduling · GPU systems · Distributed edge computing system · Real-time communication protocols

Real-Time Operating System (RTOS)

Introduction to Key OS Features

An operating system (OS) is a software component, commonly also called the kernel due its central role in modern computing systems. It serves two important functions: (1) As a hardware abstractor, it facilitates and simplifies the interactions between end-users (i.e. humans, other connected computing systems, etc.) and the hardware. (2) As a resource manager, it is responsible for efficiently distributing the hardware resources among the various software applications in the computing system so as to achieve global system-wide objectives related to efficiency and performance. These objectives could include throughput, energy consumption, responsiveness and resource utilization, among others.

Computing systems have evolved significantly since their inception in the late 1800s. In the early phase, these systems were custom-built and highly optimized to perform specific tasks (e.g. scientific computations, financial transactions, etc.). Such *batch systems* essentially executed a specific software function repeatedly on different data in a sequential manner. Their *main memory* (i.e. transient memory) layout was rather simple; a region dedicated for the customized OS and the remainder for storing data and executing the specialized function. However, their overall performance was rather limited, not only because of the lack of generalization, but also due to the fact that tasks often need to use different hardware resources at different times in their execution. So, when a task is performing some input/output (I/O) operations, the central processing unit (CPU) or processor is idling unless

the OS is able to run some other tasks on the CPU. To overcome this inefficiency, the concept of *multiprogramming* was introduced in computing systems. In a multi-programmed system, several tasks are waiting in main memory for access to various hardware resources, and the OS efficiently switches between these tasks allocating them resources depending on their needs. Although the OS functions are inherently more complex in such systems when compared to batch systems, this flexibility and dynamism leads to efficient utilization of the hardware resources.

One of the important functions of the OS is to schedule the ready tasks on the CPU. In other words, this function aims to determine which among the tasks that are in the main memory and ready to execute should be selected next for execution on the CPU. The selection strategy is of course influenced by the functional objective which could either be task-specific such as minimization of average completion time or system-wide such as throughput maximization. There are several CPU scheduling strategies including first-come-first-served (FCFS) in which tasks are prioritized based on their arrival time in the system, shortest job first (SJF) in which tasks are prioritized based on their CPU execution time and quantum-based round robin (RR) in which tasks are executed in a round robin fashion with a fixed quantum of allocated execution time. Among these scheduling strategies, RR and its variants are popular in the OS industry, mainly due to their ability to increase the responsiveness of the system; a task is guaranteed to get access to the CPU after waiting for a finite duration of time.

In a multi-programmed system, when the OS switches a task out of the CPU before it completes, it may be in the middle of executing a critical transaction comprising several instructions. For example, this transaction could be modifying a variable that is shared among several tasks or simply accessing some data that requires exclusive access. This *context switch* by the OS may lead to an erroneous scenario if the task that is switched into the CPU also accesses the same shared variable or data. The final outcome of executing these instructions from the two tasks may then depend on the specific location of the context switch, and this problem is called a *race condition*; the two tasks are racing against each other to modify the shared variable or data. *Synchronization* is a function that aims to synchronize the execution of several tasks to ensure that such race conditions do not occur. Of course this requires the task developer to annotate and identify instructions that have the potential to cause race conditions. Synchronization can be achieved either through specialized hardware instructions (e.g. TestAndSet instruction that allows tasks to atomically test and set the value of a lock variable) or through OS functions such as *semaphores*. A semaphore is a special integer variable that can only be modified by the OS, and it can be used to lock a set of task instructions so that race conditions can be prevented.

Main memory management is an important function of the OS in a multi-programmed system, because of the need to efficiently distribute this resource among the various tasks that are ready to execute. *Paging* is a popular dynamic partitioning scheme in which the entire memory region is pre-partitioned into fixed-size pages, and the OS allocates pages to tasks on demand. Thus, the number of pages allocated to a task can vary over time, enabling a very dynamic allocation

scheme that adapts to task demand at runtime. Paging also enables the concept of *virtual memory* since all the pages of a task need not be allocated in main memory at all times. Pages are brought into main memory from secondary storage on demand, which means that the memory address space accessible to a task can be much larger than the available main memory.

It is impossible to provide an in-depth coverage of the various OS functions and their interactions with hardware as well as application software in such a brief introductory note. Interested readers should refer to well-known OS textbooks (Silberschatz et al. 2018; Tanenbaum and Bos 2022).

Introduction to Real-Time Systems

Cyber-physical systems (CPS) are computing systems in which the cyber world of computation and communication is closely linked with the physical world of sensors and actuators. Such systems provide monitoring, coordination and control services for the physical counterparts and find application across a variety of domains such as avionics, automotive, medical devices, robotics, smart manufacturing, smart grids, etc. Within CPS, there exists a class of systems in which the timeliness of decisions is as important as its correctness. These are typically closed-loop control systems and deployed in safety-critical settings with stringent requirements. Examples of such systems include airbag control in automotive, flight control in avionics and robotic control systems, among several others. This class of systems is broadly referred to as *real-time systems*, in which the timeliness requirements are usually specified using *hard deadlines* for the *real-time tasks*.

As a representative example, consider the collision avoidance system in modern automotive. There are one or more sensors, such as a camera or a radar, which are observing the environment while the vehicle is in motion. The control system has to process these sensor inputs, determine the potential for a collision with an obstacle in the path and take appropriate preventive actions such as application of emergency brakes and/or steering the vehicle towards safety. The amount of time available to the system to perform these steps is strictly limited by the speed at which the vehicle is travelling and the stopping capabilities of the braking system. In other words, the control task responsible for collision avoidance must produce an output within a predetermined amount of time from the instance at which sensor data is available. This duration of time is the hard deadline imposed on the task due to system requirements and constraints.

To enable the deployment of real-time systems, the OS used must be capable of meeting the strict timing requirements. Essentially, the OS must allocate hardware resources to the real-time tasks in such a way that it ensures the satisfaction of hard deadlines under all considered circumstances. That is, predictability of meeting the timing requirements even in worst-case scenarios is the primary objective of the OS in such systems. This is in contrast to general-purpose OS discussed in section "Introduction to Key OS Features", in which the objectives are typically average-case performance metrics such as throughput maximization or completion time minimization. It is important to note that minimizing the average completion

time of tasks does not necessarily imply that the hard deadlines of tasks will be met. In order to meet such stringent timing requirements, the OS essentially has to prioritize access of the hardware resources among the tasks based on their deadlines. Furthermore, in such systems, it is also important to ensure that the time taken by various OS functions is bounded and predictable under all considered circumstances (e.g. time taken for OS functions, time to switch the CPU from one task to another, etc.). Given these specific requirements, an OS deployed in the context of a real-time system is called a *real-time operating systems* or RTOS in short form.

To meet the hard deadlines of real-time tasks, an RTOS must be aware of certain critical task parameters such as deadline, worst-case amount of execution time that the task will consume, worst-case amount of main memory required, the frequency with which tasks will be released in the system, etc. Without this information a priori, the RTOS will be unable to prioritize access to the hardware resources so as to ensure that task deadlines are always met. Task specifications in the real-time systems' literature can be broadly classified into two categories: (1) *periodic real-time tasks* in which the tasks are released into the system using a time-triggered mechanism, so that their frequency of arrival in the system can be modelled exactly using the notion of a time period (e.g. collision avoidance control system in automotive), and (2) *sporadic real-time tasks* in which the tasks are released into the system using an event-triggered mechanism, so that their frequency of arrival in the system cannot be modelled exactly (e.g. anti-lock braking system in automotive that is activated whenever the brakes are applied). To facilitate predictability, a sporadic task is additionally specified with a minimum inter-arrival time indicating the minimum separation between successive arrivals of the task. Each new arrival of a task is denoted as a *task instance* in this chapter. Example periodic and sporadic real-time tasks are illustrated in Fig. 1.

In an RTOS, although deadline-based prioritization of all the hardware resources is important, processor scheduling in particular is of prime concern and hence the focus of a substantial body of literature. This is due to the central role of the processor in ensuring that the instructions of a real-time task are executed within

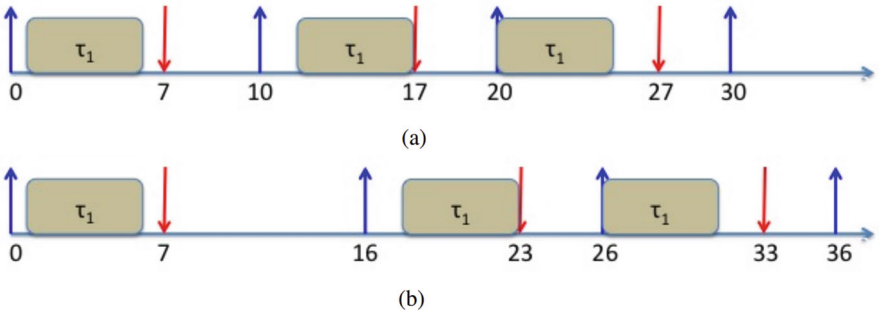


Fig. 1 Illustrating periodic and sporadic real-time tasks. The blue arrows indicate arrival times and the red arrows indicate absolute deadlines for task instances. The figure shows tasks with a period or minimum inter-arrival time value of 10 units and relative deadline of 7 units. (**a**) Periodic real-time task. (**b**) Sporadic real-time task

specified deadlines. Modern processing architectures used in real-time systems come in various flavours including single-core CPUs, multi-core CPUs with on-chip communication between the cores and the combination of CPUs and GPUs (Graphics Processing Units) with communication over the peripheral component interconnect express bus. More recently, with the advent of wireless network protocols with real-time capabilities such as 5G ultra-reliable low-latency communication (URLLC), edge computing systems in which processing servers with heterogeneous capabilities are interconnected using wireless and wired communication networks are also being envisioned for real-time systems. The remainder of this chapter focuses on each of these processing architectures in turn and introduces literature to provide insights into fundamental problems and key results.

Real-time systems is a mature area of research with extensive literature spanning more than five decades. There are also well-known textbooks with extensive coverage of some of the foundational topics in real-time systems, and interested readers can peruse them (Burns and Wellings 2009; Liu 2000; Buttazzo 2011).

Real-Time CPU Scheduling

This section focuses on literature related to CPU scheduling problems in the context of real-time systems. It first introduces key scheduling algorithms and corresponding results for single-core CPUs, followed by the same for multi-core CPUs.

Scheduling on Single-Core CPUs

Seminal contributions to the problem of single-core CPU scheduling for real-time tasks have been presented in the past (Liu and Layland 1973). This problem considers a set of real-time tasks (periodic or sporadic (For single-core CPU scheduling, no distinction is made between periodic and sporadic real-time tasks, because the worst-case arrival pattern for sporadic tasks is when the tasks arrive periodically with their minimum inter-arrival time. It has been shown that it is sufficient to consider this worst-case arrival pattern to ensure deadlines are met under all other circumstances (Dertouzos 1974; Sprunt et al. 1989; Baruah et al. 1990a). However, the same does not hold true for multi-core CPU scheduling problems.)) scheduled on a single-core CPU, and the objective is to derive prioritization strategies that can ensure the satisfaction of task deadlines. Liu and Layland (1973) introduced two categories of algorithms:

1. Under *fixed-priority* scheduling, each instance of a real-time task has the same relative priority (higher or lower) when compared to any instance of another task. In other words, priorities are fixed at the task level and do not change from one instance of the task to another. The first algorithm introduced is called the rate monotonic (RM) scheduler that prioritizes real-time tasks based on the value of their period or minimum inter-arrival time; the smaller the period or minimum inter-arrival time, the higher is the priority of the task (Liu and

Layland 1973). A further generalization of RM is the deadline monotonic (DM) scheduler (Leung and Whitehead 1982), in which tasks are prioritized based on the value of their relative deadlines (time duration between arrival of the task instance and its deadline); the smaller the relative deadline, the higher is the priority of the task. DM scheduler is equivalent to the RM scheduler whenever the relative deadline of a task is equal to its period or minimum inter-arrival time. It is worth noting that fixed-priority schedulers are extremely popular in practice (e.g. FreeRTOS (RealTimeEngineers 2003), VxWorks (WindRiverSystems 1987), QNX (Blackberry 1982), etc.), mainly because of the predictable execution of high-priority tasks and the feasibility of low-overhead implementations (e.g. bitmap based near constant runtime implementation with separate queues for each priority level).

2. Under *dynamic-priority* scheduling, each instance of a real-time task can have a different relative priority (higher or lower) when compared to another instance of a different task. In other words, priorities can change between task instances. The first algorithm introduced by Liu and Layland is called the earliest deadline first (EDF) scheduler that prioritizes task instances based on the value of their absolute deadline (actual deadline of the task instance); the earlier the deadline, the higher is the priority of the task instance. Note, the EDF scheduler is fundamentally different from the DM scheduler, because under EDF, unlike DM, an instance of task A can have lower priority than an instance of task B while at the same time have higher priority than another instance of task B.

A further generalization of dynamic-priority scheduling, in which priorities can dynamically change over time even among the same task instances, has also been proposed in the literature. Under this category, the least laxity first (LLF) scheduler is well known, and it prioritizes task instances based on the available slack time (remaining time to absolute deadline minus remaining execution time); the lower the slack, the higher is the priority of the task instance (Dertouzos and Mok 1989). Since slack can change over time, the instance priorities are also highly dynamic and change over time.

In real-time systems, ensuring that all the task deadlines are met under all considered circumstances is a fundamental requirement. It is then imperative that the performance of any CPU scheduling algorithm must be judged based on its ability to meet this requirement. *Schedulability analysis* refers to the formal analysis of any CPU scheduling algorithm to determine whether it meets this requirement. Given a collection of tasks and their specifications (deadlines, periods, worst-case execution times, etc.), the analysis aims to determine whether this taskset is *schedulable*, i.e. all task deadlines can be met assuming the specifications are valid under the given scheduling algorithm. Such an analysis facilitates in the design-time verification of the system and can also be used in formal certification processes for safety-critical applications (ISO 2018; SAE 2021).

For fixed-priority CPU scheduling, a schedulability test based on the concept of worst-case *response time* analysis has been proposed (Joseph and Pandya 1986). Worst-case response time refers to the maximum completion time for any instance of a task, and this comprises two parts: (1) the worst-case execution time of the

task instance and (2) interference from higher-priority task instances called *waiting time*. Joseph and Pandya observed that, for fixed-priority schedulers on single-core CPUs and a given real-time task, it is feasible to determine a fixed taskset release pattern that maximizes the interference from higher-priority task instances. Based on this observation, they developed an iterative closed-form equation to derive the worst-case response time for tasks. The schedulability test then involves checking that the worst-case response time of every task is no more than its deadline. This test is exact, in that it is both necessary and sufficient to ensure that all task deadlines are met. However, the runtime complexity of the test is pseudo-polynomial in the size of the input specification, because it can be proportional to the task deadline parameter. Alternatively, a polynomial-time sufficient schedulability test for RM scheduler based on the condition that the total utilization (Utilization of any real-time task is the ratio of its worst-case execution time to its period or minimum inter-arrival time.) of the taskset is no more than a specific threshold value has been proposed (Liu and Layland 1973). Finally, it has also been shown that DM is an optimal fixed-priority scheduler, in the sense that if a taskset is schedulable under any fixed-priority algorithm, then it is also schedulable under DM (Leung and Whitehead 1982).

Liu and Layland (1973) (likewise Dertouzos 1974) have shown that EDF scheduler is optimal for scheduling periodic (likewise sporadic) real-time tasks on single-core CPUs, in the sense that if a real-time taskset is schedulable then it is also schedulable under EDF. Thus, a simple polynomial-time schedulability test for EDF is to check whether the total utilization of the taskset is no more than 1. It is easy to see that any taskset whose total utilization exceeds 1 cannot be successfully scheduled on a single-core CPU, i.e. not all task deadlines can be met. However, this simple test is only applicable for tasksets in which each task's relative deadline is equal to its period or minimum inter-arrival time. For scenarios when this is not the case, a *demand bound function* -based schedulability test for EDF schedulers has been proposed (Baruah et al. 1990a,b). The demand bound function for a real-time task is the maximum total demand that all the instances of that task would generate in a given duration of time. The resulting test is exact; however, it has an exponential runtime complexity in the general case.

Scheduling on Multi-core CPUs

Scheduling real-time tasks on an *homogenous* (Cores are identical to each other.) multi-core CPU is a much harder problem when compared to single-core CPUs (Dertouzos and Mok 1989). This hardness is due to the restriction that a single task instance cannot be scheduled to run in parallel on two or more cores (This is a reasonable restriction because, in general, the code for a task instance cannot be arbitrarily parallelized.). Of course, this restriction does not prevent the scheduler from allocating different cores to the same task instance over non-overlapping time intervals. This hardness was demonstrated using a simple example taskset, which is now popularly known as the *Dhall effect* (Dhall and Liu 1978).

Fig. 2 Illustration of Dhall effect on multi-core CPU. The figure shows two cores and three tasks; τ_1 and τ_2 with utilization ϵ and τ_3 with utilization greater than $1 - \epsilon$. Due to the restriction that τ_3 cannot be scheduled in parallel on both the cores, it will miss its deadline

The taskset comprises a collection of m tasks with an arbitrarily small utilization ϵ and a single task with utilization slightly greater than $1 - \epsilon$, all scheduled on a m-core CPU. The total utilization of this taskset is arbitrarily close to 1, which means that it should become easier to schedule the taskset on the m-core CPU as m increases. However, as shown in Fig. 2, this taskset cannot be scheduled on the platform as long as each of the m tasks with ϵ utilization are scheduled on different cores, because of the restriction that the high utilization task cannot be scheduled to run in parallel. Another interesting aspect of multi-core CPUs is that the scheduling of periodic and sporadic tasks cannot be regarded as an equivalent problem. It has been shown through a simple example that there exists sporadic tasksets whose worst-case arrival pattern, in the context of multi-core CPUs, does not coincide with the scenario when all the tasks arrive periodically based on their minimum inter-arrival time (Fisher et al. 2010). In other words, for the example taskset and under a specific prioritization scheme, a deadline miss does not occur when all tasks arrive periodically with their minimum periods, but instead occurs when this is not the case.

Scheduling algorithms for multi-core CPUs can be broadly classified into two categories:

1. Under *partitioned scheduling*, real-time tasks are offline partitioned and mapped to individual processing cores, and single-core algorithms, such as those discussed in section "Scheduling on Single-Core CPUs", are used to schedule them independently on each core. This significantly simplifies the scheduling problem because all the results from single-core scheduling can now be applied to this case. However, the task to core mapping problem is a fundamentally hard problem, equivalent to the NP-complete multiple knapsack problem (Zhang and Geng 1998). Additionally, from a theoretical perspective, partitioned scheduling has a performance threshold at 50% (Oh and Bakker 1998). That is, there exist tasksets with a total utilization approximately $m/2$ which cannot be scheduled on a m-core CPU by any partitioned scheduling algorithm. A simple example of such a taskset is shown in Fig. 3; each task has a utilization slightly more than 0.5.

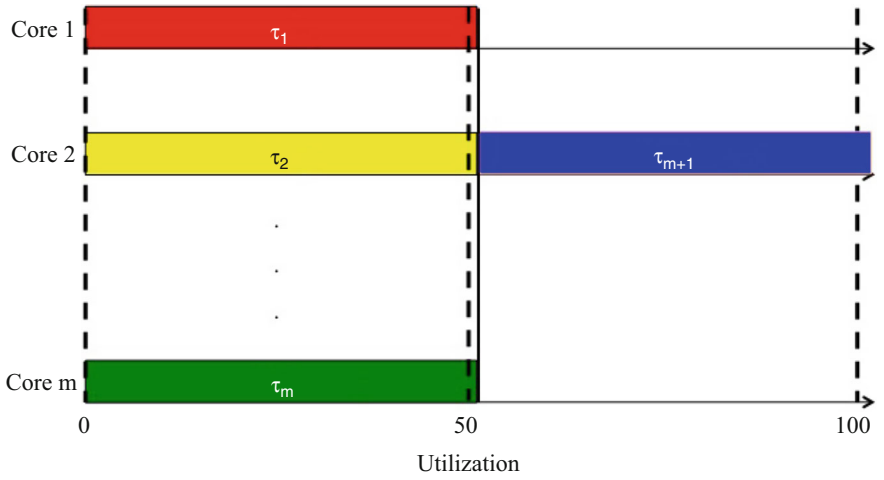


Fig. 3 Illustration of the theoretical limit of partitioned scheduling algorithms. The figure shows a m-core CPU and a collection of $m + 1$ real-time tasks. Each task τ_i has a utilization slightly more than 0.5. As can be seen, the taskset cannot be successfully partitioned on the m-core CPU

2. Under *global scheduling*, a single scheduling algorithm manages the entire multi-core CPU, and a task instance can be scheduled to run on any of the idle cores. It is also possible that a task instance has to migrate from one core to another while in the midst of its execution. This flexibility significantly enhances the capabilities of such schedulers, and in particular, they do not have a performance threshold like in the case of partitioned schedulers. However, this flexibility comes at a cost. Task migrations are in general an expensive operation in that they can significantly increase the worst-case execution time of task instances. This is because of task data in the private core-specific caches that are no longer accessible when a task instance migrates to another core. This data now has to be fetched again from lower-level shared caches or in some cases even the main memory, thus increasing the execution time of task instances.

Algorithms from section "Scheduling on Single-Core CPUs" such as RM, DM, EDF and LLF can also be used to schedule multi-core CPUs, both under partitioned and global scheduling paradigms. Under the partitioned strategy, instances of the algorithm will execute on each core and independently schedule the mapped tasks. Under the global strategy, a single instance of the algorithm will execute on one of the cores and simultaneously schedule tasks on all the cores. For example, under RM global scheduling, at any time instant, m active task instances having the lowest period values will be scheduled to run on a m-core CPU. Although all these algorithms have been extensively studied in the literature (Davis and Burns 2011), in general, their performance in terms of their ability to meet all task deadlines is largely limited. This is especially so when their performance is compared to the single-core CPU case.

The literature is abundant with studies proposing multi-core CPU scheduling algorithms that are optimal, in the sense that the algorithm can successfully schedule a taskset meeting all task deadlines if it is feasible to do so (Baruah et al. 1993; Levin et al. 2010; Regnier et al. 2011) (All these studies focus on periodic tasks with relative deadline equal to period. For sporadic tasks, it has been shown that no optimal algorithm exists unless there is clairvoyance on the future arrival of task instances (Fisher et al. 2010).). All these algorithms follow the global scheduling paradigm. A majority of them rely on the concept of *proportionate fairness*, which informally means that processing capacity must be allocated to tasks roughly in proportion to their utilization at all times. Based on different variants of this notion of fairness, a variety of optimal algorithms have been developed. However, it is important to note that achieving this fairness, and the resulting optimality, comes at a cost. These algorithms often lead to frequent task *preemptions*, that is, switching of task instances in and out of the cores, which implies potentially many task migrations. As discussed earlier, this can lead to significant increase in the worst-case execution time of task instances.

Real-Time Scheduling for CPU-GPU Systems

While CPU systems are based on the *single instruction single data* paradigm (or *multiple instruction multiple data* in the case of multi-core CPU systems) where one execution unit operates on one piece of data at a time using a single instruction, such a paradigm is not necessarily efficient at tackling *embarrassingly parallel problems* such as those found in graphics processing applications. Embarrassingly parallel problems are a class of computational problems that are easy to solve using a set of parallel operations with little or no data dependency between the parallel computations. Graphics processing units (GPUs) allow the parallelization of computations via the *single instruction multiple thread* (SIMT) parallel processing model (Vuduc and Choi 2013). Under SIMT, the same operation is performed on data stored in different locations of memory in parallel; each thread executes the same instruction, but operates on different data. SIMT enables highly parallel computation without the memory, space and power overhead of supporting completely independent compute units like traditional multi-core CPU systems. Over time, GPUs have evolved to support general purpose computational tasks as well. Such devices are referred to as *general purpose graphics processing units* (GPGPUs), and they have been used as accelerators for tasks such as computer vision, machine learning and database query processing. Such tasks often appear in applications with real-time constraints; however, GPUs' SIMT architecture introduces some unique challenges to providing real-time guarantees.

GPU Background

Early GPUs were designed specifically for graphics processing tasks and did not support general purpose computation. Through the use of *shaders*, small programs

that are run on dedicated graphics hardware, early GPUs could calculate the values for individual pixels on a display. Early shaders were inflexible and did not even support looping operations; GPUs processed shaders in a fixed pipeline starting with 3D operations like vertex and geometry shaders, and ending with pixel shaders to set the individual pixel values in the output frame buffer. Over time, however, shaders gained more functionality and GPU execution units became capable of running more general programs. Today, these small programs that run in parallel on a GPU are referred to as *kernels* to distinguish them from shaders, which are solely designed for graphical calculations.

GPU Hardware Architecture

Figure 4 shows a prototypical example of a modern GPU architecture. In practice the architectural details vary widely between GPUs from different manufacturers and the purpose is only to illustrate the key features of the SIMT architecture that lead to real-time challenges. The figure presented is an amalgamation of the popular GPU architectures developed by NVIDIA (2023), AMD (2023) and Intel (2021), but uses NVIDIA's terminology for a majority of the components due to its popularity in the literature, except for *execution units* (EUs) as CUDA core is a trademark of NVIDIA. At a high level, a GPU consists of several computational units referred to as *streaming multiprocessors* (SMs), which are roughly equivalent to AMD's

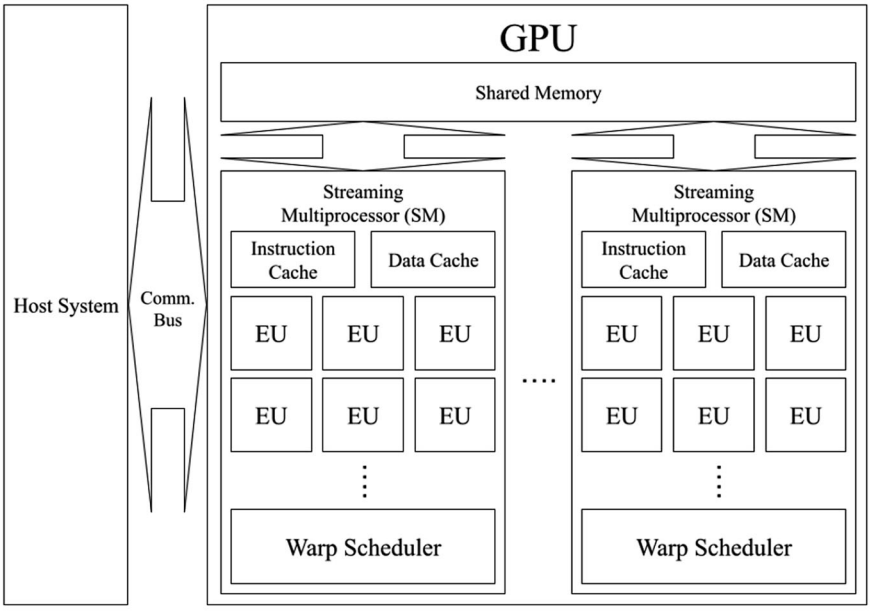


Fig. 4 A prototypical GPU architecture. The names of the components and layout will vary depending on the manufacturer. Some GPUs may contain additional cores dedicated to certain operations and different caching and memory access schemes

compute units (CUs) or Intel's slices. SMs are capable of running instructions and processing data independently of each other and have access to a large bank of shared memory where the instructions and data for processing are stored. Any new data coming from the host or data generated by the execution of a GPU kernel must be deposited in this shared memory before it can be accessed by the host system or SMs. Each SM contains its own pool of instruction memory and data memory, which acts as a cache for the slower shared memory. Different manufacturers may implement different caching schemes with additional levels of caches. Each SM contains multiple EUs (referred to as CUDA cores on NVIDIA GPUs and stream processors on AMD GPUs) which form the backbone of the SIMT architecture. Each EU in an SM executes the same instruction simultaneously, but operates on different addresses in data memory. This means, for instance, if a branch is encountered in a set of instructions, the instructions for both branch conditions must be executed on all EUs in sequence, with only the relevant EUs updating values in memory for the currently executing branch. This also implies that during looping operations, individual EUs are not freed if the loop terminates early for the data they are processing; all EUs finish processing the set of instructions simultaneously. While all the EUs in an SM must execute the same thread, multiple sets of threads can be allocated to an SM and a *warp scheduler* is used to determine which set of threads should be executing at any given time.

Some GPUs are integrated in the same package as a CPU to form a heterogeneous *system on a chip* (SoC). Internally, these integrated GPUs (iGPUs) function similar to discrete GPUs, but externally they have direct access to the same memory as their host CPU. Figure 5 illustrates a prototypical example of the memory sharing arrangement for an iGPU (Intel 2021), although the design details will vary depending on the manufacturer. In this case both the GPU and CPU share access to a last level cache (LLC), so there is no need to copy data and instructions before launching kernels on the GPU. The LLC is kept coherent with main memory by a memory controller. While this shared memory makes launching kernels on the GPU faster, the extra layers of caching can make accurately predicting response times of individual tasks a challenge.

Threading Model

From the application perspective, tasks are composed of kernels to be executed on the GPU as well as the number of *thread blocks* and threads per block that need to be executed. The execution time of a task is the time the GPU spends executing its kernel's instructions. The response time of the task is the total time spent from the host dispatching a task instance to the GPU until the time the result is received by the host. This includes the communication bus delay and the time the kernel spends blocked by other task instances waiting to execute on the GPU.

When launching a kernel as a task instance, the number of threads as well as the number of thread blocks to execute must be specified. Each thread executes the same instructions, but operates on different data. For instance when performing an operation between two vectors, the index of each element can be mapped to a thread. When performing a vector addition, the same instruction (addition) is performed

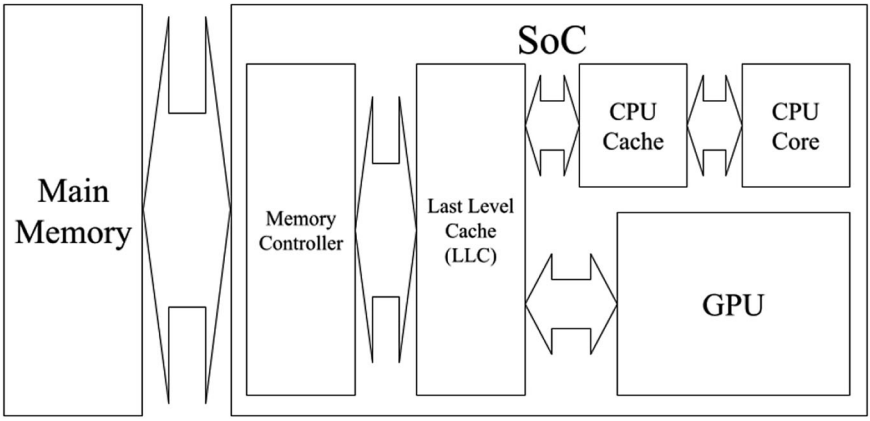


Fig. 5 A prototypical integrated GPU architecture; the GPU and CPU share access to the same pool of memory

between the corresponding indices of both vectors. However, the number of threads being executed simultaneously is limited by the resources on an SM. If a kernel is launched with more threads than available resources, groups of threads of a fixed size are executed in series until all the threads in a thread block have completed. These groups of threads are called *warps* (also referred to as wavefronts in AMD GPUs and subgroups in OpenCL). Synchronization between warps is required as there is no guarantee of the order in which they will be scheduled on a given SM.

A set of threads forms a thread block. While threads within a block can share memory, thread blocks cannot share memory among themselves. Most GPU architectures have a limit on the maximum number of threads that can be allocated to one block (e.g. 1024 threads per block in CUDA), which requires tasks operating on large data to split operations between multiple thread blocks in a process known as *tiling*. Additionally, blocks belonging to the same task cannot be synchronized, so computations in one thread block cannot be dependent on the results of computations in another thread block for the same task. When performing parallel operations, there is a trade-off between launching a kernel with a large block size and few blocks or small block size and many blocks. If the block size is small, parallel threads will not take full advantage of all the EUs in an SM, and resources will sit idle. A thread block can only be allocated to one SM, so launching a kernel as one block with the maximum number of threads will also fail to take full advantage of parallelism as the limit on the number of threads executing simultaneously will be reached, while the remaining SMs sit idle.

Scheduling Tasks on a Single GPU

Several challenges arise when trying to schedule tasks on a single GPU in a real-time system. The first is dealing with resource allocation within a single SM where

restrictions exist on how many threads can be executed simultaneously as well as the amount of instruction and data memory available to the SM. Next, there is the challenge of scheduling a set of tasks across multiple SMs in a GPU. Finally, the memory access pattern between the host and GPU affects the overall response time of GPU tasks.

Intra-SM Resource Allocation

Given a task instance to be executed as a thread block on a single SM, there are several scheduling challenges. First, while a given GPU architecture may have a maximum thread block size, the warp size dictates how many threads can execute simultaneously. This means that for thread block sizes that are not evenly divided by warp size, there will be a point where the resources on a single SM are only partially utilized, yet no other thread blocks can run. Ideally, scheduled tasks should make full use of an SM at all times.

Second, since warps from various tasks can be scheduled for execution on one SM, a warp scheduler is needed to manage which warp will execute next. By default, the hardware warp scheduler is assumed to act as a loose round robin scheduler where each warp is given the same priority and new warps are scheduled when a running warp stalls, e.g. due to a cache miss (Olmedo et al. 2020). However, this scheduling strategy can lead to many warps stalling simultaneously and cause an SM to remain idle (Narasiman et al. 2011). To combat this, a warp scheduling scheme can take advantage of memory locality like c(O)operative thread array a(W)are warp schedu(L)ing policy (OWL) (Jog et al. 2013), to reduce the time spent waiting on stalled warps. OWL also breaks the set of warps scheduled on an SM into smaller groups and prioritizes one group over the others to minimize cache contention. However, reducing idle time does not solve the problem of real-time tasks, where each task needs to complete before its deadline. For example, a task instance that is released while several other tasks are executing, but has an earlier relative deadline, may not benefit from this scheme. Another challenge with such schedulers is that, in practice, warp scheduling is achieved with dedicated hardware and off-the-shelf products may not support the ability to modify these settings.

One strategy to address this problem is to only allow a single kernel to execute on the GPU at a time and manage the queue of pending tasks on the CPU. While this means more time spent transferring kernels and data to and from the CPU, it greatly reduces variability and gives the application designer control over the order in which task instances are dispatched to the GPU. This is the idea behind the responsive GPGPU execution model (RGEM) (Kato et al. 2011a), which breaks memory accesses into smaller chunks and uses the CPU to dispatch only one kernel to the GPU at a time. Although throughput may decrease, by not using the hardware warp scheduler, variability in execution time is decreased and tasks can be dispatched to the GPU with dynamic priority.

Another issue for intra-SM scheduling is the lack of preemption. Once a kernel is launched, the process of preempting it to launch another kernel is costly, and the overhead makes preempting a running task instance on a GPU essentially impossible for real-time systems. However, due to variability in execution times, particularly when multiple tasks are scheduled on a single SM, a preemption-like

mechanism is required to prevent the possibility of priority inversion for tasks with unbounded or long-tail worst-case response times. One strategy is to break kernel executions into smaller subkernels that run sequentially and create fixed preemption points at various stages of a task's execution like the preemptive kernel model (PMK) (Basaran and Kang 2012). Under PMK, large kernels are divided into subkernels with execution times less than that of the period of higher-priority tasks. Additionally, memory accesses are broken into smaller units allowing subkernel execution and memory accesses to be interleaved. This allows the preemption of running tasks at fixed points of a scheduling window.

Inter-SM Resource Allocation

Real-time application developers have another source of control over the response time of their tasks based on how many thread blocks each kernel is launched with. Each thread block can be run on a separate SM, but multiple thread blocks from the kernel can be scheduled on the same SM if required. In general, the warp schedulers do not communicate between SMs, so another challenge is understanding which thread blocks should be executed on which resource and in what order.

A simple solution is to allocate tasks to specific SMs (Otterness and Anderson 2020). While there will never be contention between different tasks for computational resources, the individual kernels will still need to read and write from the larger block of shared memory that can be accessed by all SMs. Due to memory bandwidth and latency, it is still possible that the execution of one kernel will affect the response time of the others, leading to variability. Dealing with this issue is difficult as no method to synchronize memory access between SMs exists in most architectures.

One strategy to solve this problem is to partition the GPU's main memory to better isolate tasks allocated to individual SMs. Through reverse engineering, the fractional GPU (FGPU) framework builds on NVIDIA's multi-process service to better ensure isolation between tasks (Jain et al. 2019), by using page colouring to ensure that memory relevant to a task is allocated contiguously.

Another option is to take advantage about some knowledge of a kernel's runtime performance and co-locate kernels to the same SM that complement each other. One example of this is the ISPA framework (Zhao et al. 2023) that uses elastic thread block sizes to decide which kernels to co-locate in order to take advantage of tensor cores and CUDA cores in the same SM.

Finally, given knowledge about the tasks sharing a GPU, a dynamic scheduling algorithm can be created that can deal with unknown release times and multiple tasks vying for a limited number of GPU resources. Unlike the standard multi-core CPU scheduling problem, the tasks for a GPU can be dynamically resized and split. For example, if a kernel is launched with a blocksize of 1024, by keeping track of the associated memory, this can be split at runtime and several kernels of smaller blocksize can be launched. This allows a kernel that would have previously been limited to execution on one SM to be split across several. By taking into account all tasks in a taskset, these resized kernels can be allocated to different priority queues

upon release and scheduled for execution across the GPU's SMs as shown in the Slate framework (Allen et al. 2019).

All the techniques described here rely on first identifying the execution characteristics of a kernel on a given device offline. Without this knowledge the best one can do is to allocate specific tasks to specific SMs. In particular, it is not just the average response time characteristics that need to be known, but the worst-case ones. This is especially true as two tasks can interact adversely with each other depending on their location in memory and resource demands. One strategy to determine this is to purposefully create "enemies" for a given task that seek to maximize interference (Yandrofski et al. 2022).

Memory Transfer Between Device and Host

Another major impediment to real-time performance on GPUs is the time spent transferring data between the device and host. If this communication occurs over a bus that is shared with other devices, response times of GPU tasks may no longer be bounded as other devices on the bus could block it indefinitely. Even if the GPU is the only device, there is still the issue of ensuring that the time spent dispatching newly released task instances to the GPU does not interfere with the ability to retrieve data from currently running task instances. RGEM (Kato et al. 2011a) attempts to address this issue by breaking memory accesses into smaller pieces to reduce the amount of blocking.

Even in shared-memory architectures, memory access time is an issue. For instance, a GPU task may require more memory than is physically available to the system. In such cases, memory from a currently executing task will have to be paged to disk and other memory pages retrieved. In general, there is no way to place a bound on response time when paging occurs, but works have analysed the possibility of paging from the GPU directly to solid state non-volatile memory to decrease paging time. Due to the speeds of modern solid-state memory, such a paging scheme can drastically reduce paging time and allow reduction in response times of tasks on an oversubscribed GPU (Bakita and Anderson 2022).

Another issue that can occur is when freeing and allocating memory on a GPU causes the host system to block. Such an issue was identified in some NVIDIA GPUs and can be detected with the CUDA Pitfall Detector for Real-Time Systems (CUPiD^{RT}) library (Amert and Anderson 2021). Even if tasks executed on the GPU do not have real-time constraints, this can affect the real-time performance of tasks that remain on the host CPU. Careful analysis of GPU tasks is required to ensure that a CPU-GPU system will not experience this problem and mitigation must be taken if necessary.

Multi-GPU and CPU-GPU Scheduling

Not all real-time systems are limited to a single GPU. In such systems it is necessary to understand how to schedule a task set across multiple GPUs, given their unique

execution properties. Furthermore, GPUs are often part of *heterogeneous systems*, systems that contain several classes of computational resources with differing execution properties. This leads to additional challenges and opportunities in the context of real-time systems.

Multiple GPUs Controlled by One Host

Several challenges arise when scheduling a taskset across multiple GPUs where all tasks are launched from the same host. From a real-time scheduling perspective, once a task instance is launched on a GPU, it cannot be preempted. Furthermore, task instances executing on individual GPUs do not have global access to memory, meaning any dependent tasks need to either run subsequently on the same GPU or wait for the data to be offloaded back to the host and transferred to another GPU. One strategy to address this involves breaking tasks into smaller units called codelets, which consist of an implementation on each resource available to the system, their data dependencies and their outputs (Augonnet and Namyst 2009). These codelets can then be scheduled across the available resources taking into account the time for offloading and transferring data between the different GPUs.

Another solution is to use a data-aware scheduler that considers the location of the output of a task's precedence constraints in memory and uses this to determine on which GPU a task should run. PTask (Rossbach et al. 2011) accomplishes this for parallel tasks on a system with multiple GPUs. Furthermore, either fixed-priority or dynamic-priority schedules can be generated, depending on the needs of the system. By considering precedence constraints, PTask enables a dataflow-style programming interface where the output of one task is used as the input to another. This style of programming is natural for many types of real-time GPU applications that involve processing data in a pipeline, e.g. gesture recognition; being able to handle these constraints is very important.

Similar to time sharing in single-core CPU systems, resources can also be partitioned in time and space to enforce isolation in GPU systems. TimeWall (Amert et al. 2021) can provide this partitioning for real-time GPU systems by using global and in-partition schedulers, as well as a special locking protocol that enforces time partition boundaries.

Heterogeneous Systems as DAGs

Because of the importance of precedence constraints in many real-time applications, it is natural to model certain real-time workloads as *directed acyclic graphs* (DAGs). Figure 6 illustrates such a DAG where each vertex corresponds to an instance of a task and unidirectional edges between vertices correspond to precedence constraints. Because this is a heterogeneous system, some task instances will only be able to execute on certain computational resources, e.g. inferencing a neural network may be reserved for execution on a GPU, while if-then rules may be reserved for execution on a CPU. In Fig. 6 a task's resource affinity is represented by the shape of each vertex (Lin et al. 2022). The tuples associated with each vertex carry additional information about the task such as its index (0th element) and its worst-case execution time (1st element).

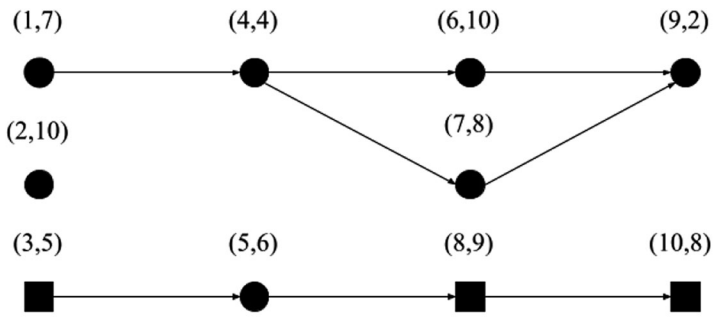


Fig. 6 An example of a DAG. Different shaped vertices correspond to tasks that must be run on a particular type of device. Edges indicate precedence constraints, while the tuple above each vertex is in the format (*task index*, *worst-case execution time*)

Once a taskset has been modelled as a DAG, schedulability analysis can be performed for a given set of heterogeneous resources and scheduler. It has been shown that the schedulability of a DAG can be represented as a zero-one integer linear program (Baruah 2020). Furthermore, scheduling such a DAG is possible with federated scheduling (Lin et al. 2022), where each task is assigned one of four execution modes: exclusive allocation, semi-exclusive allocation, sequential or shared.

Splitting Tasks Between CPUs and GPUs

Systems with both CPUs and GPUs can take advantage of the fact that both resources are capable of executing the same tasks, but at different rates and with different numerical precision. When a DAG is used to model a system, the assumption is that one task is mapped to one resource because that resource is required to obtain the correct functional result; however, if the application can tolerate slightly different results depending on where the task is executed, this provides the opportunity to meet deadlines while minimizing idle hardware time. For example, on a GPU, a neural network is executed with 32-bit floating point precision, while on a CPU, a quantized version of the same neural network can be executed. While the functional results of both models will be slightly different, the overall accuracy may still be acceptable for certain applications. However, instead of just running one task solely on the GPU and another solely on the CPU, there is also the possibility to offload tasks from the GPU to CPU and vice versa during their execution (Kang et al. 2021). For example, in a neural network, some layers may yield a better functional result on one resource than the other, and task instances can be migrated to that resource provided there is enough time remaining for offloading. Analysis can be performed on a given task set to show whether or not it is schedulable while still meeting its required accuracy bounds. Additionally, as the tasks execute, any slack, i.e. time remaining until a task's deadline minus the task's remaining execution time, can be reclaimed by the system and used to increase accuracy beyond that of the initial schedule.

Application Domains

GPUs appear in a number of computing domains and face real-time data processing challenges in each one. The following sections analyse specific challenges in some common application domains; however, this list is not exhaustive.

Graphics Processing

GPUs were originally designed for the task of graphics processing, which includes calculating geometric transforms and pixel shading for the rendering of 3D graphics. Examples of "soft" real-time tasks in this domain are multitasking windowing systems and high-performance 3D graphics for video games. These tasks are characterized by a need for low response times, but can still tolerate frame drops. Dealing with fairness in multitasking graphics systems is one area where GPU task scheduling is important (Kato et al. 2011b). Additionally, two applications may be generating graphical output for the GPU to process and this data could be bursty, i.e. task release times are aperiodic and arrive more frequently in some time intervals than others. In some graphics systems, hard real-time deadlines are required (Zou et al. 2023), for example, the instrument cluster of a vehicle. In such cases, graphical tasks need to be divided across a GPU's available SMs and schedulability analysis must be performed to show that the workload is schedulable. There is also a trade-off between being able to meet hard deadlines and maximizing the GPU's throughput; the schedule that maximizes throughput does not necessarily allow all tasks to meet their deadlines. Some frameworks like TimeGraph Kato et al. (2011b) try to allow an application to have the best of both worlds by providing two operating modes that a system can dynamically switch between.

Cloud Systems

Offloading data processing to the cloud is used to increase operational efficiency and remove the risks of maintaining local hardware for performance computation. As GPUs can be used for massively parallel computations, it is natural to maintain GPU clusters in a cloud environment, but such clusters introduce their own challenges. Firstly, large computations require coordination between multiple GPUs introducing the same scheduling problems as multi-GPU systems. Additionally, in the cloud environment, multiple virtual machines (VMs) may be allocated to share a single GPU. While it is possible to simply partition a GPUs resources in time and space (parallel SMs), this can lead to severely underutilized hardware if all running VMs are not using their allocations all the time. To improve upon this scheme, oversubscription can be allowed (Yao et al. 2023), which allows a VM to dynamically use more than its share of resources if conditions permit. Additionally, interference between scheduled tasks on different VMs can lead to deadline violations of other tasks, and frameworks have been proposed to deal with this problem in the context of cloud GPUs (Xu et al. 2023). In the cloud, not every problem requires hard real-time constraints. Sometimes guaranteeing *quality of service* (QoS), a probabilistic measure of the availability of a resource, is enough.

While such systems do not need to meet every deadline, checks are still needed to ensure that a given QoS can be maintained given the taskset allocated to a particular GPU system.

Tools and Frameworks

While many scheduling techniques and analyses are applicable to GPUs in general, there are some caveats for different GPU hardware and software frameworks. The following section introduces some of the current mainstream frameworks and their applicability to real-time systems.

NVIDIA and CUDA

Currently, NVIDIA GPUs are widely used in research and industrial applications. The CUDA framework abstracts away the hardware details of GPU programming and enables kernels to be written that are compatible across NVIDIA GPU families (NVIDIA 2023). Newer GPUs also include tensor cores co-located on SMs with traditional CUDA cores that are optimized directly for matrix multiplication. NVIDIA's embedded GPUs feature direct memory access; however, their multi-process framework is not currently supported on such GPUs. So additional steps need to be taken when attempting to run multiple real-time tasks on NVIDIA SoCs. Additionally, the implementation details about the hardware and device drivers are not always known, so a lot of work has gone into reverse engineering models of the underlying components, although they are not always accurate (Olmedo et al. 2020).

AMD and ROCm

AMD offers an API similar to CUDA for their GPUs called ROCm. In AMD's terminology, a CU is roughly equivalent to an SM, a streaming processor is roughly equivalent to a CUDA core, and a wavefront is roughly equivalent to a warp (AMD 2023). ROCm is entirely open source (Otterness and Anderson 2020), which opens up many possibilities for real-time application development by allowing modifications of scheduling algorithms without having to reverse engineer what the manufacturer is doing.

OpenCL

OpenCL is an API maintained by several manufacturers with a focus on portability across devices (AMD 2010). It supports GPUs from Intel, AMD, NVIDIA, Qualcomm and others as well as field programmable gate array (FPGA) devices as parallel compute units. In OpenCL terminology, a CU is roughly equivalent to an SM, a processing element is roughly equivalent to an EU, and a subgroup is equivalent to a warp. Like ROCm, OpenCL is also open source, but instead emphasizes portability over performance on one particular platform.

Alternative Architectures

While so far this section focused on presenting the challenges of real-time scheduling with a generic SIMT architecture, there are several variations on this architecture that are worth mentioning, as they can also affect the real-time performance of a CPU-GPU system.

Processing in Memory

A major challenge with all GPU architectures is the amount of time spent transferring data to and from memory. This is partly due to the physical distance of the GPU computational units from the on-chip memory. Processing in memory (PIM) attempts to solve this problem by placing low-power GPU cores physically close to memory, so that they can operate with low delay on data stored in memory, while a high-power GPU sits far from memory, but is capable of processing more data in parallel at higher speeds (Pattnaik et al. 2016). If PIM becomes widely used, it will face several challenges with regard to real-time performance. Firstly, tasks need to be scheduled not only in time but also allocated to the low-power or high-power cores. These task allocations may be dynamic, depending on the nature of the system. Secondly, PIM seeks to reduce energy usage by placing cores physically close to memory. Finding a schedule for a taskset that can minimize energy usage while still meeting deadlines is equally important.

FPGAs as Accelerators

Another alternative to GPUs is to use FPGAs as accelerators for parallel tasks. FPGAs allow a designer to create their own custom digital logic, which may be better suited for certain real-time tasks than a generic GPU. Additionally, GPU caching and memory access schemes make deterministic estimates of response times very difficult, whereas on an FPGA, the system designer has full control over these design attributes. In terms of real-time execution, OpenCL already provides several execution modes unique to FPGA devices that can improve performance under certain conditions (Jiang et al. 2020).

Real-Time Edge Computing Systems

Introduction to Edge Computing

Edge computing is a distributed computing paradigm that allows devices to execute their tasks at the "edge" of the network. Distributed computing refers to executing tasks across multiple systems or devices interconnected by a wired and/or wireless network as shown in Fig. 7. Each system has its own computation resource (CPUs and/or GPUs), memory (storage) and communication resource. A distributed system typically follows a peer-to-peer architecture where a device or system acts as both server and clients. They can either request for resources or share their resources

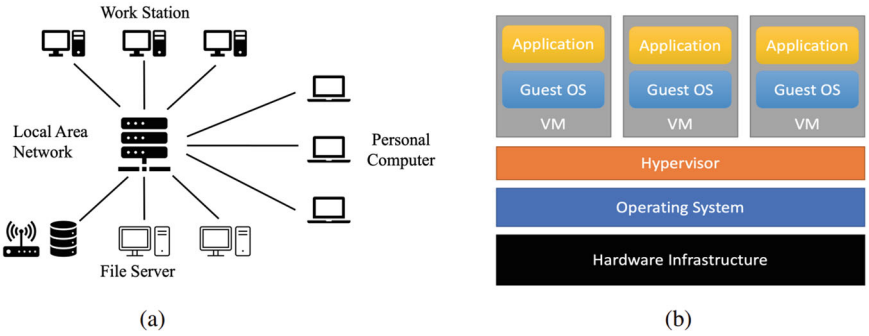


Fig. 7 Various distributed computing architectures. (**a**) Sample distributed system. (**b**) Virtualization

among devices in the network to jointly perform the computation at a much higher rate. The key characteristics of a distributed system include resource sharing, scalability, concurrency, fault tolerance and heterogeneity. With the advent of *virtualization*, a hardware abstraction process, distributed systems evolved into high-performance heterogeneous servers. Virtualization refers to the process of creating a virtual layer over the hardware platform to create multiple computing instances (virtual machines) and execute them simultaneously on the same hardware platform (host machine).

The advancement in the internet technology made task execution and storage on remote servers feasible with reasonable response times. This has led to the development of cloud-based applications with service-level agreements that guarantee a minimum quality of service. The resource allocation in conventional cloud computing can broadly be classified as *reservation-based* and *on-demand* provisioning. The resources are charged based on the hardware configuration (CPUs, GPUs and memory), type of provisioning (reservation/on-demand) and the duration of provisioning (hours/days) (Armbrust et al. 2009). Most common objective in cloud resource provisioning is the cost and/or energy minimization. Cloud computing provides resource migrations between servers and replications as a load balancing and fault-tolerance mechanism, respectively. Fundamentally, the tasks in cloud computing transmit their data to a centralized server for execution. However, growing complexity and unprecedented scale of data from a multitude of distributed edge devices—sensors, Internet-of-things (IoT) devices such as smart cameras, medical devices—there is a tremendous burden on the network capabilities of the remote servers.

In order to address this concern, edge computing was introduced wherein the task execution happens at the edge of the network. Performing the computation closer to the devices minimizes the response time of tasks as the delay incurred in transmitting the data to the servers is greatly reduced. For instance, modern applications such as autonomous driving or industrial automation need to transmit huge amount

of raw data for processing. Such applications require efficient computation and faster response times for reliable operation. Unlike conventional cloud servers, edge servers are located closer to the device which enable them to transmit and process the data at a faster and reliable rate. By allowing storage and execution capabilities closer to the end devices, edge computing improves the system performance and also reduces the computation and bandwidth requirement of the devices. Moreover, the latest 5G network can provide high-speed low-latency communication with massive network capacity enabling near-real-time analysis and response for edge devices. Thus, edge computing is an innate choice for designing modern real-time applications.

The Edge Architecture

The edge resources are modelled as either a set of servers or a tiered network of servers interconnected by a low-latency, large bandwidth and high-speed backhaul network for data communication. Internally, the edge servers may be connected to a larger resource capacity cloud servers using a core network. The edge servers comprise an access point through which the end devices that are within the coverage area can access it. A sample edge architecture is shown in Fig. 8.

Computation Resources The edge servers may have multiple hardware resource types such as processors (both CPUs and GPUs), memories (RAM) and storage (HDDs and SSDs) or software services such as AI/ML modules, databases, etc. Depending on the nature of application and the type of problem being addressed, the

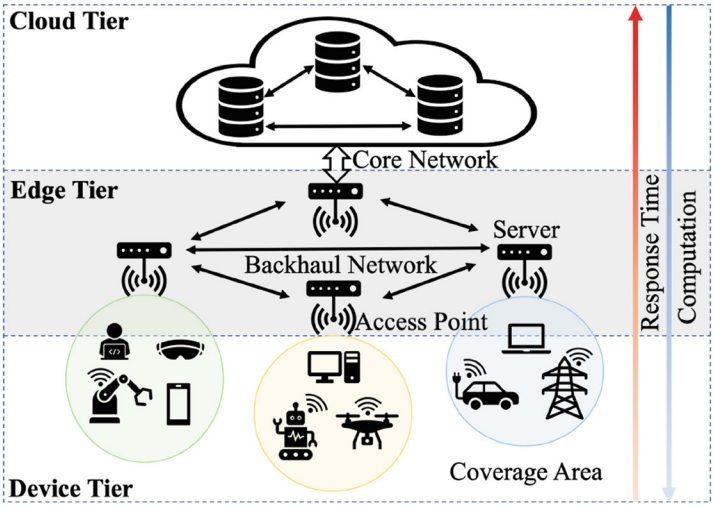


Fig. 8 A tiered edge-cloud architecture. Response time and computation resources increase as one goes higher up

term "computation resource" may refer to any of these resources. For instance, the most widely interpreted resource in this context is the virtual machine (VM) with a certain configuration of processors, memory and storage. Applications may request for a certain number of VMs with a specific configuration for a specified (sometimes unspecified) duration of time. Some studies also abstract the computation resource as processing cycles per second. Devices may send their (local) data, execute it using the resources at the edge layer and get the result (processed data) back to the devices.

Communication Resources Besides the computation resources used by the devices for execution, edge network also comprises of resources for communication. These resources are utilized by the devices mainly to establish a communication between the edge servers and transmit their data back and forth. The devices in the coverage area of the edge access points can request the servers for bandwidth allocation. Often end devices are allocated a certain bandwidth for communication and data transfer to the servers. The bandwidth of the servers (including gateways, routers and switches) determines the amount of devices (or the number of tasks) that can be connected to it for data transfer. In case of resource unavailability at one server, the tasks (or VMs) can be offloaded (or migrated) to another server for execution if bandwidth is available for communication. Typically, the uplink (or offloading) bandwidth is more constrained than the downlink (or downloading) bandwidth due to the amount of data to be transmitted by the devices.

Real-Time Edge Computing

The advent of URLLC in 5G has made edge computing a reality for many real-time applications such as autonomous multi-agent systems, connected cars, digital twins, etc. To enable the deployment of such real-time systems on edge servers, the OS used must manage the edge resources, both computation and communication, more efficiently. Essentially, the OS must allocate resources such that the timing constraints of the tasks are adhered. In addition, the OS must be capable of handling time synchronization of devices and servers to facilitate correct schedule of operation. Owing to the wide variety of devices connected to the network, ensuring connectivity and maintaining a synchronous notion of time is a challenging task. Real-time edge computing typically involves three stages: (i) *computation offloading*, (ii) *resource provisioning* and (iii) *task scheduling*. First, computation offloading refers to the decision problem of whether to offload the task from the end device to the edge servers or not. Second, resource provisioning refers to the allocation problem of communication (bandwidth) and computation resources (multiple-type resources). Finally, task scheduling problem similar to the CPU scheduling determines when to execute the task and on which server.

The following presents timing parameters widely adopted in real-time edge computing and the factors that constitute it. The *execution time* of an offloading task depends on the amount of computation resource allocated to it on the edge server.

The execution time may also include any queuing delay due to the task buffers implemented at these servers. The *queuing delay* is the waiting time experienced by the task due to some pending tasks arrived ahead at the servers. The *communication time* is the time required to transmit the data from one entity (device/server) to another. It depends on the amount of bandwidth allocated to it and the size of data to be transmitted. The communication time includes the delay incurred in the transmission medium, due to several factors such as signal strength, noise level, interference and distance. Since the servers are internally connected by a high-speed network, the delay incurred in transferring the data between servers is much smaller compared to the delay in transmitting the data from/to the end devices.

Timing Model Generally, it is assumed that each task (from the device) requests edge resources for a certain duration of time and may have some timing constraints associated with it. The *response time* of an offloading task is given by the summation of execution time and communication time. The literature in real-time edge computing focuses on either of the two following timing model:

- Constrained Deadline: The works that bound the response time of the task (hard real-time tasks) are classified under this category.
- Response Time Minimization: The remaining works are categorized here. This includes the body of work that minimizes the average task response time or overall makespan. The *makespan* is defined as the largest task response time.

Resource Allocation in Real-Time Edge

This section focuses on literature related to edge resource scheduling problems in the context of real-time systems. It first introduces resource contention models and presents a simple classification based on resource type. Then, it describes some assumptions on the edge architecture from the literature. Next, it briefly introduces some key timing related model parameters that contribute to the task response time. Owing to the growing popularity of edge systems, there is a plethora of work in the context of real-time systems. It would not be possible to cover the entire literature into these fixed classes. Throughout, the section tries to elucidate the different real-time resource allocation/scheduling techniques applied on each stage using representative works.

Contention Model

Edge resources, although abundant with respect to the end devices, are constrained by the capacity in general. When multiple tasks are hosted on the edge server, they tend to have contention between other tasks co-hosted on the same edge server. Depending on the type of resource these tasks contend, the contention model of the edge server is broadly classified into:

1. Under *no contention*, both the communication and the computation resources are not shared by the tasks. In other words, fixed resources are allocated to the

tasks and the total resources allocated are lower than the capacity of the servers. Works in this category mainly focus on the task offloading (Kao et al. 2017), server provisioning (Chen and Xu 2019) and/or task scheduling problems (Guo et al. 2017) with optimization objectives such as minimizing task response times (Kao et al. 2017; Chen and Xu 2019) and device energy (Guo et al. 2017). A greedy task replication strategy for fault tolerance and a multi-arm bandit learning algorithm based on a probabilistic prediction of task response time is proposed (Chen and Xu 2019). The server provisioning for tasks and task scheduling problem is modelled as a collection of trees with end-to-end deadlines and fixed offloading bandwidth (Guo et al. 2017).

2. Under *computation contention*, the tasks contend only for the computation resources in general and it is usually bounded by the computation capacity of the edge servers. Works in this category mainly focus on the server provisioning and task scheduling problems with optimization objectives such as minimizing the task response times (Dai et al. 2018; Ren et al. 2017), device energy (Yaqub and Sorour 2018) and server energy (Chen et al. 2019). The joint task offloading and server provisioning problem with fixed offloading bandwidth for tasks is considered, where the offloading problem is solved using bipartite graph-based rounding method and the provisioning problem is solved using gradient descent method (Dai et al. 2018). The same problem is solved for partial offloading using convex optimization (Ren et al. 2017). A priority-based heuristic and bisection method for offloading decisions and Lagrangian method for provisioning problem is considered in Yaqub and Sorour (2018). The switching costs and energy loss trade-offs for server activations and deactivations are solved using Vickrey auctions (Chen et al. 2019).
3. Under *communication contention*, the tasks contend only for the communication resources in general and it is usually bounded by the bandwidth capacity of the edge servers. Works in this category mainly focus on task offloading and server/bandwidth provisioning problems with optimization objectives such as minimizing server energy (Sun et al. 2017), device energy (Chen et al. 2015), server usage costs and communication overhead (Yu et al. 2018). A probability function for the task deadline misses is derived considering the bound on queuing delay (Sun et al. 2017). The bandwidth is modelled as a function of interference among tasks and the problem is modelled in a decentralized game-theoretic framework and solved using potential games. The problem in Yu et al. (2018) is formulated as a multi-commodity max-flow problem and solved.
4. Under *computation and communication contention* both the communication and the computation resources are in general shared by the tasks, both of which are bounded by their respective capacities. Works in this category mainly focus on the task offloading, server provisioning and/or task scheduling problems with optimization objectives such as minimizing task response times (Castellano et al. 2019; Heydari et al. 2019; Jošilo and Dán 2019), makespan (Pang et al. 2017) and minimizing VM delays (Cziva et al. 2018). The task offloading problem is formulated as a Markov decision process and actor-critic-based reinforcement learning heuristic to learn the offloading decisions is proposed in

Heydari et al. (2019). A decentralized max-consensus-based greedy algorithm for the combined task offloading and server provisioning problem is presented (Castellano et al. 2019). The same problem is modelled in a decentralized game-theoretic framework and solved using Stackelberg games (Jošilo and Dán 2019). A decentralized heuristic using dynamic programming is also developed for the problem (Pang et al. 2017).

The studies covered in this section are developed for single-tier architecture comprising a set of servers. Similar set of studies can be found in the literature targeted at multi-tier architecture.

Tiered Architecture

The edge-cloud architecture comprises a multi-layer set of servers interconnected by a high-speed network. Depending on the assumption of server architecture, the resource allocation problem dimension changes. For instance, in the case of computation offloading, the decision problem evolves into consideration of offloading at the edge or the cloud layer. Besides the offloading of tasks from the devices, there is also offloading of tasks from edge servers (lower tier) to cloud servers (higher tier). Likewise, the resource provisioning and task scheduling need to consider prominent factors such as server-to-server delay and migration costs between servers. Generally, the multi-layer servers are implemented with task buffers and adopt a service rate approach for handling task execution as opposed to conventional resource provisioning due to the difference in resource type and variation in execution time across the server layers. The following presents some studies distinct to multi-tier architecture based on the contention model previously defined.

The server provisioning problem on multi-tier architecture with both computation and communication contention is formulated as a pure integer non-linear programming (PINLP) and solved iteratively using a solver in Gao et al. (2019). A lazy switch algorithm to control the task migration frequency between servers is also proposed. Considering no contention and constrained deadline model, a singleton-weighted congestion game-based heuristic was developed to arrive at a consensus on task allocation at the lower tier (Zhang and Wang 2019). A stochastic Lyapunov optimization-based greedy heuristic is also considered to estimate task response times and decide whether to provision the task on another server at the higher tier. The task offloading and server provisioning problem is formulated as a mixed integer non-linear programming (MINLP), and a branch and bound algorithm is designed to find an optimal solution and prune the search space (Vu et al. 2018).

The task scheduling problem with only computation contention is formulated as a MINLP. The response time is modelled using an M/M/1 queuing model where M stands for Markovian and one server and solve the optimization problem using heuristic solution by decomposing the problem into sub-problems (Zeng et al. 2016). An online learning algorithm for joint task offloading and server provisioning using multi-arm bandit with a parameterized regret bound is presented (Ouyang et al. 2019). The server provisioning and task scheduling problem with a fixed

offloading bandwidth per task and fractional resource allocations is solved optimally using convex optimization and branch-and-bound methods (Tong et al. 2016). For the same problem, a decentralized solution by selecting the server with the least increase in response time and schedule using the shortest remaining computation time first policy is developed (Tan et al. 2017).

Model Parameters

Several factors attribute to the task response times in edge computing. The three most explicitly used notions of time in edge computing are the execution time, deadline and communication time. The execution time or resource reservation time denotes the fixed duration of time required to execute the task (or use the resource) on the servers. The deadline denotes the bound on the task response time. Several deadline constrained studies provide this parameter. The communication time is considered by those studies that consider an offloading bandwidth for tasks (fixed/variable). Besides these three explicit parameters, two parameters indirectly contribute to the task response time under the contention model. The computation capacity of the edge servers affects the execution time of the task, whereas the communication capacity of the edge server plays a role in determining the communication time. This section introduces two model parameters: service rate and transmission delay. The former refers to the rate at which the tasks are served (executed) at the server. The latter denotes the additional network delay in the transmission medium of the system.

1. The *service rate* and arrival rate of the tasks are considered by relatively fewer studies in the real-time edge literature (Gao et al. 2019; Xiao and Krunz 2017). Most works use a Markovian queuing model, either M/M/m or M/M/1, where m or 1 denotes the number of servers. Under this model, the arrival rate of the tasks is characterized by a Poisson process and the service time follows an exponential distribution.
2. The *transmission delay* in a multi-tier architecture consists of device-server delay (Gao et al. 2019; Ouyang et al. 2019; Tan et al. 2017; Vu et al. 2018; Xiao and Krunz 2017) and server-server delay (Ouyang et al. 2019; Vu et al. 2018; Xiao and Krunz 2017). The device-server delay is the time taken by the device/server to transmit the data vice versa in excess of data transmission time. This parameter plays a vital role in determining the server for task offloading. Studies in the literature had modelled it as a constant as well as a variable parameter. In practice, this translates to the server distance from the device or the signal strength of access point used for communication. The server-server delay parameter is usually introduced when dealing with task offloading and migrations between servers, that is, when tasks are offloaded from one server to another server, which may or may not be located at the same tier. The task offloading and server resource provisioning are the two problems pertaining to this parameter.

The response time of tasks executing in edge servers depends on both the computation time and the communication time. Most resource allocation studies

in real-time edge either do not consider the communication delay or abstract the communication delay between the devices and/or servers using timing parameters. Guaranteeing end-to-end deadlines in edge computing is not feasible without incorporating the delay parameters for the communication. Several real-time communication algorithms specific to a communication medium have been designed. The following section discusses the algorithms and protocols designed for real-time networks.

Introduction to Real-Time Networks

Real-time networks are communication networks where each communication is deterministic and needs to be executed within a hard deadline. This deadline is pre-specified depending on the application requirement. A real-time distributed system such as autonomous vehicles, industrial automation, smart grids, etc. can be represented by an edge-cloud architecture (refer to Fig. 8 in Section "The Edge Architecture") where a large number of real-time tasks are either executed on the edge devices or offloaded to the cloud or edge server. These systems are associated with three types of communications—(1) communications between the edge devices, (2) communications between an edge device and the cloud or edge server and (3) communications between the cloud or edge servers. For example, the collision avoidance application in autonomous vehicles is a real-time application where the basic safety messages (BSM) are transmitted periodically between the edge devices, i.e. the vehicles. Failure to deliver these messages within deadline may lead to accident of the vehicle. Similarly, in industrial automation systems, a large number of sensors which serve as the edge devices are deployed across the factory floor. The sensor data is transferred periodically to the remote controller (the cloud or the edge server) which generates the control signals to actuate the system. Since these applications are time-critical in nature, each of these communications is a real-time communication, i.e. associated with a hard deadline. Thus, real-time networks are widely adopted as the communication media in these systems. The network protocols used for these communications need to guarantee the message delivery without violating the deadlines of the systems. Real-time networks can be either wired or wireless depending on the application and the platform on which it is deployed.

Real-Time Wired Networks

Most of the communications in automobiles and industrial automation systems are still dominated by wired connectivity. For example, real-time wired networks are adopted in automobiles to facilitate communication between the electronic component units (ECUs) that are installed on the edge devices, i.e. the vehicles. Real-time wired networks are also prevalent in most industrial automation systems to ensure uninterrupted, high-performance connectivity among the edge devices

(sensors, actuators, motors) and the cloud or edge servers (controllers). Some of the wired network protocols that are widely adopted for real-time communications include the control area network (CAN) bus communication networks (Othman et al. 2006), FlexRay networks (Makowitz and Temple 2006), TTEthernet (Kopetz et al. 2005), etc. A brief overview of each of these protocols is presented below.

Control Area Network (CAN) A *CAN* bus is a simple, robust, low-cost and easily accessible vehicle bus standard designed for communication among the electronic devices embedded in a vehicle (Othman et al. 2006). Prior to the introduction of the CAN bus, each electronic device in a car was connected to every other device. With increase in the implementation of different functions in automobiles, it was very difficult to maintain the complex wiring system. This led the automotive industry to introduce the CAN bus system that allows the ECUs to communicate with each other by connecting each ECU to the common serial bus, thereby reducing the wiring overheads of the system. Although the CAN bus has been developed for automobiles, nowadays, it is also used to connect some of the vital components such as the microcontrollers and the sensors in industries.

The CAN protocol defines a set of rules to transmit and receive messages through the serial bus. These network devices are known as nodes. The nodes are equipped with specific hardware and software to enable transmission and reception of messages. Whenever a node has some data to transmit, it checks the state of the CAN bus. If the bus is idle, the node writes the data frame on the bus. All the nodes on the shared bus receive the data frame. This data frame does not contain the addresses of the transmitting or receiving node. Instead, a unique arbitration ID labels each data frame. Depending on this ID, the CAN node decides whether to accept or reject the frame. If multiple nodes attempt to transmit data at the same time, the node with the lowest arbitration ID gets access to transmit. Thus, nodes with higher arbitration ID have to wait until the bus becomes available for data transmission. It has been shown both theoretically and experimentally that the worst-case response time in a CAN bus is bounded (Tindell et al. 2000). As a result, the communications among the CAN nodes are deterministic. This ensures predictable data transmissions with real-time guarantees in a CAN network.

FlexRay Although the CAN bus communication network is flexible and highly scalable, it has limited bandwidth, i.e. the arbitration process in a CAN bus does not allow high data rate, and hence, it cannot always guarantee timely delivery of data. To overcome these drawbacks, *FlexRay* network protocol was introduced (Makowitz and Temple 2006). The FlexRay is the first time-triggered network protocol that is capable of handling both deterministic data arriving at a specific time frame in a predictable manner and event-triggered data.

The communication in a FlexRay network is time-division multiple access (TDMA) based, i.e. time is divided into slots and each communication occurs in such slots. This TDMA based communication in a FlexRay network guarantees determinism and ensures timely delivery of data. Hence, unlike CAN network, FlexRay is very suitable for real-time applications. The duration of a time-slot

depends on the application requirement. All the nodes in a FlexRay network are synchronized to the same clock. The time-slots for communication are assigned to the nodes when they join the network. A node can write to the bus only in the slots assigned to it.

The communications in a FlexRay network occur in cycles, the duration of which varies between 1 and 5 ms. A communication cycle is divided into four segments:

1. A **static segment** transmitting deterministic messages
2. A **dynamic segment** transmitting event-triggered messages
3. A **symbol window** to transmit network initialization signals and network maintenance messages
4. An **idle time** to synchronize between the node clocks

TTEthernet Although FlexRay has support for both deterministic real-time applications and event triggered non-real-time applications, it is not scalable for large complex networks. Thus, *Time-Triggered Ethernet (TTEthernet)* was specifically designed to support complex networks in large-scale real-time applications such as aerospace, automobiles, etc (Kopetz et al. 2005). TTEthernet supports deterministic real-time communications over the Ethernet providing much more scalability as compared to CAN or FlexRay networks.

TTEthernet is fully compatible with IEEE 802.3 Ethernet standards (IEEE 2018). In a TTEthernet network, the time-triggered traffic always takes precedence over standard Ethernet traffic. Thus, TTEthernet guarantees very precise individual packet level scheduling which makes it very suitable for safety-critical applications. In addition, TTEthernet also supports packet replication, i.e. each packet is re-transmitted immediately through another channel if the channel in use becomes faulty. This helps to overcome failures ensuring reliable communications.

Real-Time Wireless Networks

To facilitate large-scale infra-structures, the number of devices in most real-time systems keep increasing. With deployment of large-scale infra-structures, the device connectivity gets denser day by day. This, in turn, results in an increase in the wiring overhead in these systems. Thus, most large-scale real-time systems are switching from wired to wireless communication. In addition to this, wireless communication also provides flexibility in deployment by enabling frequent changes in the network topology depending on the link quality. Thus, real-time wireless networks are widely adopted for communications in large-scale industrial control plants, unmanned aerial vehicles, vehicular networks, etc. A brief overview of some of the wireless protocols that have real-time capabilities is presented below.

Bluetooth *Bluetooth* is a simple, low-power, low-cost wireless technology supporting short range communication (McDermott-Wells 2005). It is based on wireless personal area network (WPAN) technology which is used to convey messages

over short distances (approximately 10 m) among a group of participating devices. Initiation of a connection through Bluetooth is quite straightforward and requires almost no infrastructure. Hence, any device can join or leave the network at any point in time. The Bluetooth-enabled devices operate at 2.4 GHz ISM band and use a spread spectrum, frequency hopping and full-duplex signal that hops among 79 frequencies in a pseudo-random manner. The architecture and operations in Bluetooth-enabled devices are based on IEEE 802.15.1 standards. The Bluetooth-enabled network devices connect and communicate with each other in their radio proximity and dynamically establishes ad-hoc networks known as piconets. Each piconet can accommodate a maximum of 8 Bluetooth-enabled devices, i.e. 1 master device and 7 slave devices, and has a coverage of up to 10 m.

Although Bluetooth is highly suitable for the implementation of industrial wireless sensor networks (WSNs), it cannot guarantee real-time message transmissions with bounded delays. To overcome this limitation, a real-time stack on the top of Bluetooth known as *RESEMBLE*, has been proposed (Leonardi et al. 2023). RESEMBLE has provisions to support both real-time and non-real-time communications on the same network. Additionally, RESEMBLE also provides clock synchronization, enabling efficient TDMA-based communications that are necessary for real-time transmissions with hard deadlines in industrial control applications.

IEEE 802.11 Although RESEMBLE in Bluetooth protocol has provisions to support both real-time and non-real-time applications, the communication in Bluetooth is very short range. Thus, Bluetooth is not suitable for long-range real-time communications such as vehicular communications. To support long range communications, *IEEE 802.11*, more commonly known as the *Wifi* standard, has been used (Crow et al. 1997). IEEE 802.11 is based on the architecture and specifications of wireless local area networks (WLANs). There are several standards of IEEE 802.11 WLANs, such as 802.11n (Nee et al. 2006) used in multiple input multiple output (MIMO) antennae, 802.11p (Arena et al. 2020) used in vehicular environments to support intelligent transportation systems, etc. All of these standards under 802.11 use carrier-sense multiple access with collision-avoidance (CSMA/CA) for communication,

With the application of IEEE 802.11 wireless technology in control systems, the flexibility in deployment and maintenance of the network increases. However, IEEE 802.11 cannot provide real-time guarantees on the message delivery of the system. Additionally, IEEE 802.11 does not satisfy the high frequency of message transmissions which is required in most control applications. To overcome these shortcomings, a real-time high-speed wireless communication, more specifically *real-time Wifi* or *RT-Wifi* was designed (Wei et al. 2013). RT-Wifi is a TDMA-based protocol designed on the physical layer of IEEE 802.11 with high sampling rate (up to 6 kHz) and deterministic timing guarantees on message delivery. The architecture for RT-Wifi is designed in such a way that it has provision to support both real-time and non-real-time communications in the same network.

ZigBee The *ZigBee* wireless protocol is designed to transmit small amounts of data over a short distance with very low power consumption (Safaric and Malaric 2006). ZigBee is based on low-rate WPAN (LR-WPAN) technology. It is much simpler compared to Bluetooth and is based on IEEE 802.15.4 standards (IEEE 2016). It is highly suitable and widely adopted in smart manufacturing, home automation, internet-of-things (IoT), WSNs, etc., for its ease of installation, reliable data transfer, short-range communication, low-cost devices and reasonable battery life.

Two different types of devices are used in ZigBee networks:

- A full functional device (FFD) that can be used either as a PAN coordinator, or as a coordinator, or as a device
- A reduced functional device (RFD) that can be used in very simple application

A ZigBee network must include at least one FFD to serve as the PAN coordinator. The PAN coordinator serves as the primary controller of the PAN and is responsible for initiating, terminating and routing the communication in the network. Based on the functions associated with the devices, there are three different types of devices in a ZigBee network: ZigBee Coordinator, ZigBee Router and ZigBee End Device. A ZigBee network is associated with only one ZigBee Coordinator. However, the ZigBee network can have multiple ZigBee Routers and multiple ZigBee end devices.

ZigBee networks are secure and reliable in nature. It supports 128-bit advanced encryption standard (AES) encryption. The communications in ZigBee are CSMA/CA based. However, to support low-latency requirements of real-time applications, the devices in ZigBee can also transmit in some guaranteed time-slots. The schedule for communications in these guaranteed time-slots is pre-computed in order to satisfy the deadlines of real-time applications.

WirelessHART The *WirelessHART* is the most suitable and reliable WSN protocol that satisfies the hard deadlines and guarantees collision-free deterministic communications in real-time systems (Chen et al. 2010). It is based on IEEE 802.15.4 standards and is the most widely adopted wireless protocol in industrial control systems. A WirelessHART network consists of a gateway, multiple field devices, access points (APs) and a centralized network manager. The network manager, installed on the gateway, manages the devices, creates the routes, optimizes the network and generates the transmission schedule for the devices. The field devices are mainly wireless sensors and actuators. The APs, connected to the gateway, generate redundant paths between the gateway and the field devices for reliable communication.

The key characteristic features of a WirelessHART network are as follows.

1. **TDMA based communication:** The communication in a WirelessHART network is TDMA based with time-slots of 10 ms, during which a network device sends a message and receives its corresponding acknowledgement.
2. **Channel diversity:** A WirelessHART network operates at 2.4 GHz ISM band. The entire frequency band is divided into 16 channels, defined in IEEE 802.15.4

physical layer. In order to avoid interference from neighbouring wireless systems, the WirelessHART adopts channel hopping, i.e. the operating frequency of the communication channel changes in every time-slot. Additionally, if a channel suffers from persistent external interference, then that channel is blacklisted and not used for communication.

3. **Route Diversity:** To mitigate physical obstacles, broken links, interference, etc., the WirelessHART allows transmission of a packet via multiple paths over different channels.
4. **Spatial re-use of channels:** The WirelessHART network schedules transmissions based on multi-channel TDMA protocol. In large networks deployed over a wide area, two distant nodes can use the same channel simultaneously if they do not interfere with each other.

All these characteristics of WirelessHART ensure reliable and predictable communications which make it very suitable for hard real-time systems.

Discussion Existing WSNs are not sufficient to support the ultra-low-latency requirement (in milliseconds or sub-milliseconds) for many futuristic applications such as remote surgery, connected cars, autonomous vehicles, smart manufacturing, etc. 5G networks are going to dominate the future wireless connectivity. The URLLC service in 5G networks with ultra-low latency (1 ms) and very high reliability (99.99%) is designed specifically to support these type of modern applications with real-time requirements in the future WSNs (Ali et al. 2021). However, the deployment of URLLC in real-time systems is still in its infancy. Hence, the real-time capabilities of URLLC are not elaborated upon in this chapter.

Real-Time Flow

This subsection introduces some formal notations and terminologies to explain a real-time flow and a feasible schedule for real-time flows.

Based on the features of a real-time WSN, a real-time network can be modelled as a network graph, $G = (V, E)$, where V is the set of nodes, which are the network devices and the APs, and E is the set of edges or links between the nodes or the network devices. A node $v \in V$ can be either a sensor device, an actuator device or an AP. An edge $e = u \rightarrow v$ is part of G if and only if node u can reliably communicate with node v. In a transmission along an edge $u \rightarrow v$, node u is the sender and node v is the receiver of the transmission. Two transmissions along edges $u \rightarrow v$ and $w \rightarrow x$, where $u, v, w, x \in V$, are said to be conflicting transmissions if both of them have the same sender or the same receiver, i.e., if $(u == v) \vee (v == w) \vee (u == x) \vee (v == x)$.

A typical real-time WSN consists of n periodic real-time flows, $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$, defined over a network graph, $G = (V, E)$, with m channels. Each flow $F_i \in \mathcal{F}$ generates a packet at the source node $s_i \in V$ every p_i time-slots. The packet needs to be delivered via route R_i to the destination node d_i, where

$d_i \in V \setminus \{s_i\}$. This packet delivery needs to be completed within relative deadline δ_i from its release, where $\delta_i \leq p_i$. Since the same network graph is shared by multiple flows, these flows can have overlapping routes, i.e. an edge can be shared by multiple flows in the network.

The following terminologies are used to explain a feasible schedule for a set of real-time flows.

Release Time The **release time** of the jth instance of flow F_i, $(j \geq 1)$ is the time at which the jth instance of F_i is released at the source node s_i. The release time (r_{ij}) is defined as

$$r_{ij} = (k-1) \times p_i \tag{1}$$

Number of hops The **number of hops** in the route of a flow F_i is the number of transmissions required to reach the destination node (d_i) from the source node (s_i).

Scheduling Window **Scheduling window** (W_{ij}) of the jth instance of the ith flow F_i is the time-slots between its release time r_{ij} and deadline δ_{ij}. Time-slot $\sigma \in W_{ij}$, if $r_{ij} + 1 \leq \sigma \leq \delta_{ij}$.

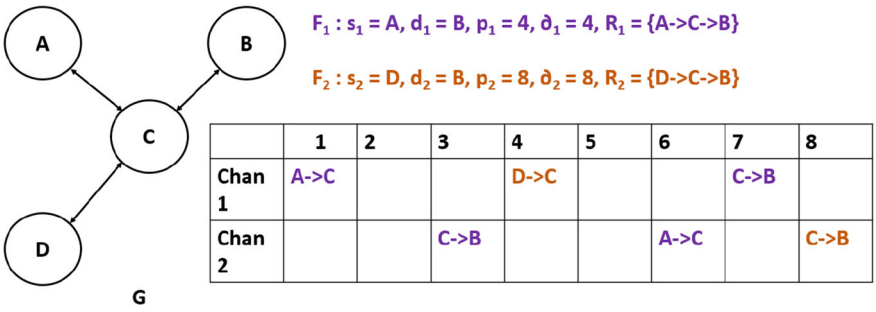
Feasible Schedule Given a network graph $G = (V, E)$ with m channels and a set of n flows, $\mathcal{F}$, a feasible schedule $\mathcal{S}$ is a sequence of transmissions over the time-slots in $\mathcal{S}$ (Samaddar et al. 2020). Each transmission is a mapping of a flow to a channel in a time-slot satisfying the following conditions:

1. **No transmission conflict:** Two transmissions along $u \rightarrow v$ and $w \rightarrow x$ are scheduled in the same time-slot t, if $u \rightarrow v$ and $w \rightarrow x$ are non-conflicting transmissions.
2. **No collision:** If $u \rightarrow v$ uses channel y and $w \rightarrow x$ uses channel z in the same time-slot t, where $u, v, w, x \in V$, and u, v, w, x lie within the same interference region in G, then $y \neq z, \forall y, z \in [1, m]$.
3. **No deadline violation:** If a flow F_i has h hops, $(F_i \in \mathcal{F})$, then all h hops of F_i are to be scheduled within its deadline δ_i.
4. **Flow sequence preservation:** If a flow F_i has h hops, $(F_i \in \mathcal{F})$, then the kth hop $(1 < k \leq h)$ cannot be scheduled until all the previous $k-1$ hops are scheduled.

The length of a feasible schedule is given by the number of time-slots in the feasible schedule $\mathcal{S}$ and is equal to the length of the hyperperiod, i.e. the lowest common multiple of the periods of the real-time flows in the system.

Any real-time flow scheduling algorithm needs to follow the above four constraints, i.e. no transmission conflicts, no collision, no deadline violation of flows and flow sequence preservation, to schedule a real-time flow.

Figure 9 shows a feasible schedule $\mathcal{S}$ over the network graph G with two periodic real-time flows F_1 (marked in purple) and F_2 (marked in orange) and with two channels. The flows, F_1 and F_2, have their sources at node A and node D,



	1	2	3	4	5	6	7	8
Chan 1	A->C			D->C			C->B	
Chan 2			C->B			A->C		C->B

Fig. 9 A feasible schedule $\mathcal{S}$ over a network graph G with two periodic real-time flows, F_1 (in purple) and F_2 (in orange) and two channels. The flows have their sources $s_1 = A$, $s_2 = D$; destinations, $d_1 = d_2 = B$; period $p_1 = 4$ time-slots, $p_2 = 8$ time-slots; relative deadline $\delta_1 = 4$ time-slots, $\delta_2 = 8$ time-slots; routes $R_1 = \{A \rightarrow C \rightarrow B\}$, $R_2 = \{D \rightarrow C \rightarrow B\}$

respectively. Both of them have their destination at node B. The periods and the relative deadlines of F_1 and F_2 are 4 time-slots and 8 time-slots, respectively. Since the period of the flows is equal to their relative deadlines, the flows have implicit deadlines. The hyperperiod of the schedule $\mathcal{S}$ is 8 time-slots. In this hyperperiod schedule $\mathcal{S}$, F_1 has two flow instances with release time at time-slot 0 and 4 and F_2 has one flow instance with release time at 0. Therefore, the scheduling windows associated with the two instances of F_1 are given by the time-slots in [1,4] and [5,8], respectively. Similarly, the scheduling window associated with the instance of F_2 is given by the time-slots in [1,8]. Since both the flows require two transmissions to reach the destination node B from their respective source nodes, the number of hops for both them is 2.

Routing and Scheduling in Real-Time Wireless Sensor Networks

Routing is defined as the process of selecting paths for the data packets in the network that are flowing from the source device to the destination device with proper network traffic management. Routing of data packets in WSNs is challenging due to several restrictions in the WSNs such as node deployment, limited energy consumption, scalability, fault tolerance, network heterogeneity, network connectivity, etc. Based on these factors, some routing algorithms exist in the literature such as dynamic source routing (DSR) (Johnson and Maltz 1996), Ad-hoc On-Demand Distance Vector Routing (AODV) (Perkins and Royer 1999), greedy forwarding (GF) (Takagi and Kleinrock 1984), etc. However, none of these algorithms satisfy the timeliness requirements of real-time systems. Hence, they are not applicable for routing real-time flows in the network. The main objective of a routing algorithm in a real-time system is to route the real-time flows in the best possible way so that it can guarantee the deadlines of the real-time flows while still satisfying the transmission delay bound in the network. The following subsections discuss two popular real-time routing protocols in WSNs.

RAP Routing Protocol

The RAP routing protocol is considered to be the first real-time communication architecture for large-scale WSNs (Lu et al. 2002). It is designed for CSMA-based communication in real-time WSNs. It comprises a network stack that consists of a location-addressed protocol, a geographic forwarding protocol, a velocity monotonic scheduling (VMS) and a prioritized MAC to support the WSN communication. Each of these network components runs a set of efficient algorithms to reduce the deadline miss ratio of the real-time flows in the WSN. The central layers in the RAP capture the real-time constraints of the WSN. The RAP uses VMS that calculates priority of a data packet based on two parameters: (1) the distance between the current node and the destination node and (2) the deadline associated with the data packet. Thereafter, it adjusts the inter-frame spaces based on the priority of the data packet during its wait time and back-off time.

SPEED Routing Protocol

The SPEED routing protocol is an important real-time routing protocol from which many other routing protocols such as MMSPEED, FTSPEED, etc., have been derived (He et al. 2003). Although the RAP routing protocol handles real-time traffic by locally degrading a certain proportion of the non-real-time traffic, this type of local MAC layer adaptation in RAP cannot handle long-term congestion in hotspot areas. Hence, the SPEED routing protocol is designed that provides a combination of MAC layer and network layer adaptation to handle these issues. The SPEED is a QoS routing protocol that guarantees soft real-time deadlines of the flows. The SPEED protocol uses velocity to choose the route of a flow. It measures the velocities of the links and considers those links that satisfy the required velocity as the forwarding paths. Since no adjustment is done in the MAC layer, the velocity depends entirely on the link quality and network traffic at runtime. The SPEED protocol has a backpressure scheme that re-routes a packet along links with larger delays if the current transmission link gets congested.

Routing and scheduling protocols for real-time WSNs exist in the literature in abundance (Saifullah et al. 2010; Yu et al. 2013; Zheng et al. 2017). However, most of them are designed mainly for specific wireless standards, e.g. WirelessHART, and are based on some specific routing metric, such as least laxity first (Saifullah et al. 2010), QoS (Zheng et al. 2017), packet reception ratio (Kim 2021), etc. Thus, they cannot be generalized for all types of real-time WSNs. Interested readers should refer to survey papers on routing and scheduling algorithms for different wireless standards in the literature (Nobre et al. 2015; Kim et al. 2017).

Summary

A wide variety of real-time applications are emerging in the fields robotics, automotive, healthcare and smart manufacturing. To guarantee predictability and timeliness, real-time operating systems are usually deployed on computing platforms used by these systems. This chapter has explained the characteristics of scheduling in

the context of real-time systems for various computing architectures including the distributed edge servers and communication networks.

The discussion on different computing architectures has shown that the scheduling algorithm and protocols for real-time systems has to consider the implications of architectural features, execution models and the network topology. In recent years, more real-time applications are getting built on distributed edge servers comprising traditional CPUs, GPUs and network. A separate discussion on resource provisioning schemes for each architecture has shed some insights on achieving both computation and communication with real-time guarantees.

References

Ali R, Zikria YB, Bashir AK, Garg S, Kim HS (2021) Urllc for 5G and beyond: Requirements, enabling incumbent technologies and network intelligence. IEEE Access 9:67064–67095. https://doi.org/10.1109/ACCESS.2021.3073806

Allen T, Feng X, Ge R (2019) Slate: enabling workload-aware efficient multiprocessing for modern GPGPUs. In: 2019 IEEE international parallel and distributed processing symposium (IPDPS). IEEE, pp 252–261. https://doi.org/10.1109/IPDPS.2019.00035

AMD (2010) Introduction to OpenCL programming. AMD, Santa Clara

AMD (2023) Introducing AMD CDNA 2 architecture: propelling humanity's foremost research with the world's most powerful HPC and AI accelerator. AMD, Santa Clara

Amert T, Anderson JH (2021) Cupidrt: detecting improper GPU usage in real-time applications. In: 2021 IEEE 24th international symposium on real-time distributed computing (ISORC). IEEE, pp 86–95. https://doi.org/10.1109/ISORC52013.2021.00022

Amert T, Tong Z, Voronov S, Bakita J, Smith FD, Anderson JH (2021) Timewall: enabling time partitioning for real-time multicore+accelerator platforms. In: 2021 IEEE real-time systems symposium (RTSS). IEEE, pp 455–468. https://doi.org/10.1109/RTSS52674.2021.00048

Arena F, Pau G, Severino A (2020) A review on ieee 802.11p for intelligent transportation systems. J Sens Actuator Netw 9(2). https://doi.org/10.3390/jsan9020022

Armbrust M, Fox A, Griffith R, Joseph AD, Katz RH, Konwinski A, Lee G, Patterson DA, Rabkin AS, Stoica I, Zaharia MA (2009) Above the clouds: a Berkeley view of cloud computing. Science 53:07–013

Augonnet C, Namyst R (2009) A unified runtime system for heterogeneous multi-core architectures. In: César E, Alexander M, Streit A, Träff JL, Cérin C, Knüpfer A, Kranzlmüller D, Jha S (eds) Euro-Par 2008 workshops – parallel processing. Springer, Berlin/Heidelberg, pp 174–183

Bakita J, Anderson JH (2022) Enabling GPU memory oversubscription via transparent paging to an NVMe SSD. In: 2022 IEEE real-time systems symposium (RTSS). IEEE, pp 370–382. https://doi.org/10.1109/RTSS55097.2022.00039

Baruah S (2020) Scheduling dags when processor assignments are specified. In: Proceedings of the 28th international conference on real-time networks and systems, RTNS'20. Association for Computing Machinery, New York, pp 111–116. https://doi.org/10.1145/3394810.3394813

Baruah S, Mok A, Rosier L (1990a) Preemptively scheduling hard-real-time sporadic tasks on one processor. In: Proceedings of IEEE real-time systems symposium, pp 182–190. https://doi.org/10.1109/REAL.1990.128746

Baruah S, Rosier L, Howell R (1990b) Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. Real-Time Syst 2:301–324. https://doi.org/10.1007/BF01995675

Baruah SK, Cohen NK, Plaxton CG, Varvel DA (1993) Proportionate progress: a notion of fairness in resource allocation. In: Proceedings of the ACM symposium on theory of computing, pp 345–354

Basaran C, Kang KD (2012) Supporting preemptive task executions and memory copies in GPGPUs. In: 2012 24th Euromicro conference on real-time systems. IEEE, pp 287–296. https://doi.org/10.1109/ECRTS.2012.15

Blackberry (1982) Blackberry QNX. https://blackberry.qnx.com/en. Accessed: 03 Aug 2023

Burns A, Wellings A (2009) Real-time systems and programming languages, 4th edn. Addison Wesley Longmain. https://www.cs.york.ac.uk/rts/books/RTSBookFourthEdition.html

Buttazzo GC (2011) Hard real-time computing systems: predictable scheduling algorithms and applications, 3rd edn. Springer Publishing Company, Incorporated

Castellano G, Esposito F, Risso F (2019) A distributed orchestration algorithm for edge computing resources with guarantees. In: IEEE INFOCOM 2019-IEEE conference on computer communications. IEEE, pp 2548–2556

Chen D, Nixon M, Mok A (2010) WirelessHART: real-time mesh network for industrial automation. Springer. https://doi.org/10.1007/978-1-4419-6047-4

Chen L, Xu J (2019) Task replication for vehicular cloud: contextual combinatorial bandit with delayed feedback. In: IEEE INFOCOM 2019 – IEEE conference on computer communications. IEEE Press, pp 748–756. https://doi.org/10.1109/INFOCOM.2019.8737654

Chen S, Jiao L, Wang L, Liu F (2019) An online market mechanism for edge emergency demand response via cloudlet control. In: IEEE INFOCOM 2019-IEEE conference on computer communications. IEEE, pp 2566–2574

Chen X, Jiao L, Li W, Fu X (2015) Efficient multi-user computation offloading for mobile-edge cloud computing. IEEE/ACM Trans Netw 24(5):2795–2808

Crow B, Widjaja I, Kim J, Sakai P (1997) IEEE 802.11 wireless local area networks. IEEE Commun Mag 35(9):116–126, https://doi.org/10.1109/35.620533

Cziva R, Anagnostopoulos C, Pezaros DP (2018) Dynamic, latency-optimal VNF placement at the network edge. In: IEEE infocom 2018-IEEE conference on computer communications. IEEE, pp 693–701

Dai Y, Xu D, Maharjan S, Zhang Y (2018) Joint offloading and resource allocation in vehicular edge computing and networks. In: 2018 IEEE global communications conference (GLOBECOM). IEEE, pp 1–7

Davis RI, Burns A (2011) A survey of hard real-time scheduling for multiprocessor systems. ACM Comput Surv (CSUR) 43(4):1–44

Dertouzos M, Mok A (1989) Multiprocessor online scheduling of hard-real-time tasks. IEEE Trans Softw Eng 15(12):1497–1506. https://doi.org/10.1109/32.58762

Dertouzos ML (1974) Control robotics: the procedural control of physical processes. Inf Process 74:807–813

Dhall SK, Liu CL (1978) On a real-time scheduling problem. Oper Res 26(1):127–140. https://doi.org/10.1287/opre.26.1.127

Fisher N, Goossens J, Baruah S (2010) Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible. Real-Time Syst 45:26–71. https://doi.org/10.1007/s11241-010-9092-7

Gao B, Zhou Z, Liu F, Xu F (2019) Winning at the starting line: joint network selection and service placement for mobile edge computing. In: IEEE INFOCOM 2019-IEEE conference on computer communications. IEEE, pp 1459–1467

Guo J, Song Z, Cui Y, Liu Z, Ji Y (2017) Energy-efficient resource allocation for multi-user mobile edge computing. In: GLOBECOM 2017–2017 IEEE global communications conference. IEEE, pp 1–7

He T, Stankovic J, Lu C, Abdelzaher T (2003) Speed: a stateless protocol for real-time communication in sensor networks. In: 23rd international conference on distributed computing systems, 2003. Proceedings, pp 46–55. https://doi.org/10.1109/ICDCS.2003.1203451

Heydari J, Ganapathy V, Shah M (2019) Dynamic task offloading in multi-agent mobile edge computing networks. In: 2019 IEEE global communications conference (GLOBECOM). IEEE, pp 1–6

IEEE (2016) IEEE standard for low-rate wireless networks. IEEE Std 802154-2015 (Revision of IEEE Std 802154-2011), pp 1–709. https://doi.org/10.1109/IEEESTD.2016.7460875

IEEE (2018) IEEE Standard for Ethernet. IEEE Std 8023-2018 (Revision of IEEE Std 8023-2015). pp 1–5600. https://doi.org/10.1109/IEEESTD.2018.8457469

Intel (2021) Intel processor graphics gen11 architecture. Intel, Santa Clara

ISO (2018) Iso26262: Road vehicles – functional safety. https://www.iso.org/standard/68383.html. Accessed: 03 Aug 2023

Jain S, Baek I, Wang S, Rajkumar R (2019) Fractional GPUs: software-based compute and memory bandwidth reservation for GPUs. In: 2019 IEEE real-time and embedded technology and applications symposium (RTAS). IEEE, pp 29–41. https://doi.org/10.1109/RTAS.2019.00011

Jiang J, Wang Z, Liu X, Gómez-Luna J, Guan N, Deng Q, Zhang W, Mutlu O (2020) Boyi: a systematic framework for automatically deciding the right execution model of OpenCL applications on FPGAs. In: Proceedings of the 2020 ACM/SIGDA international symposium on field-programmable gate arrays, FPGA'20. Association for Computing Machinery, New York, pp 299–309. https://doi.org/10.1145/3373087.3375313

Jog A, Kayiran O, Chidambaram Nachiappan N, Mishra AK, Kandemir MT, Mutlu O, Iyer R, Das CR (2013) Owl: cooperative thread array aware scheduling techniques for improving GPGPU performance. SIGPLAN Not 48(4):395–406. https://doi.org/10.1145/2499368.2451158

Johnson DB, Maltz DA (1996) Dynamic source routing in ad hoc wireless networks. Springer, Boston, pp 153–181. https://doi.org/10.1007/978-0-585-29603-6_5

Jošilo S, Dán G (2019) Wireless and computing resource allocation for selfish computation offloading in edge computing. In: IEEE INFOCOM 2019-IEEE conference on computer communications. IEEE, pp 2467–2475

Kang W, Lee K, Lee J, Shin I, Chwa HS (2021) Lalarand: flexible layer-by-layer CPU/GPU scheduling for real-time DNN tasks. In: 2021 IEEE real-time systems symposium (RTSS). IEEE, pp 329–341. https://doi.org/10.1109/RTSS52674.2021.00038

Kao YH, Krishnamachari B, Ra MR, Bai F (2017) Hermes: latency optimal task assignment for resource-constrained mobile computing. IEEE Trans Mob Comput 16(11):3056–3069

Kato S, Lakshmanan K, Kumar A, Kelkar M, Ishikawa Y, Rajkumar R (2011a) RGEM: a responsive GPGPU execution model for runtime engines. In: 2011 IEEE 32nd real-time systems symposium. IEEE, pp 57–66. https://doi.org/10.1109/RTSS.2011.13

Kato S, Lakshmanan K, Rajkumar R, Ishikawa Y (2011b) TimeGraph: GPU scheduling for real-time multi-tasking environments. In: 2011 USENIX annual technical conference (USENIX ATC 11). USENIX Association, Portland. https://www.usenix.org/conference/usenixatc11/timegraph-gpu-scheduling-real-time-multi-tasking-environments

Kim BS, Park H, Kim KH, Godfrey D, Kim KI (2017) A survey on real-time communications in wireless sensor networks. Wirel Commun Mob Comput 2017:1–14. https://doi.org/10.1155/2017/1864847

Kim MK (2021) Efficient link scheduling based on estimated number of packets in queue on industrial wireless sensor networks. Energies 14(19). https://doi.org/10.3390/en14196370, https://www.mdpi.com/1996-1073/14/19/6370

Kopetz H, Ademaj A, Grillinger P, Steinhammer K (2005) The time-triggered ethernet (TTE) design. In: Eighth IEEE international symposium on object-oriented real-time distributed computing (ISORC'05), pp 22–33. https://doi.org/10.1109/ISORC.2005.56

Leonardi L, Lo Bello L, Patti G (2023) Resemble: a real-time stack for synchronized mesh mobile bluetooth low energy networks. Appl Syst Innov 6. https://doi.org/10.3390/asi6010019

Leung JYT, Whitehead J (1982) On the complexity of fixed-priority scheduling of periodic, real-time tasks. Perform Eval 2(4):237–250. http://dblp.uni-trier.de/db/journals/pe/pe2.html#LeungW82

Levin G, Funk S, Sadowski C, Pye I, Brandt S (2010) Dp-fair: a simple model for understanding optimal multiprocessor scheduling. In: Euromicro conference on real-time systems. IEEE, pp 3–13

Lin CC, Shi J, Ueter N, Günzel M, Reineke J, Chen JJ (2022) Type-aware federated scheduling for typed DAG tasks on heterogeneous multicore platforms. IEEE Trans Comput 72(5):1286–1300. https://doi.org/10.1109/TC.2022.3202748

Liu CL, Layland JW (1973) Scheduling algorithms for multiprogramming in a hard-real-time environment. J ACM 20(1):46–61. https://doi.org/10.1145/321738.321743

Liu JWS (2000) Real-time systems. Prentice Hall, Upper Saddle River. http://www.amazon.com/Real-Time-Systems-Jane-W-Liu/dp/0130996513

Lu C, Blum B, Abdelzaher T, Stankovic J, He T (2002) Rap: a real-time communication architecture for large-scale wireless sensor networks. In: Proceedings. Eighth IEEE real-Time and embedded technology and applications symposium, pp 55–66. https://doi.org/10.1109/RTTAS.2002.1137381

Joseph M, Pandya P (1986) Finding response times in a real-time system. Comput J 29:390–395. https://doi.org/10.1093/comjnl/29.5.390

Makowitz R, Temple C (2006) Flexray – a communication network for automotive control systems. In: 2006 IEEE international workshop on factory communication systems, pp 207–212. https://doi.org/10.1109/WFCS.2006.1704153

McDermott-Wells P (2005) What is bluetooth? IEEE Potentials 23(5):33–35. https://doi.org/10.1109/MP.2005.1368913

Narasiman V, Shebanow M, Lee CJ, Miftakhutdinov R, Mutlu O, Patt YN (2011) Improving GPU performance via large warps and two-level warp scheduling. In: Proceedings of the 44th annual IEEE/ACM international symposium on microarchitecture, MICRO-44. Association for Computing Machinery, New York, pp 308–317. https://doi.org/10.1145/2155620.2155656

Nee RV, Jones VK, Awater GA, van Zelst A, Gardner J, Steele G (2006) The 802.11n MIMO-OFDM standard for wireless LAN and beyond. Wirel Pers Commun 37:445–453

Nobre M, Silva I, Guedes LA (2015) Routing and scheduling algorithms for wirelesshartnetworks: a survey. Sensors 15(5):9703–9740. https://doi.org/10.3390/s150509703, https://www.mdpi.com/1424-8220/15/5/9703

NVIDIA (2023) CUDA C++ programming guide. NVIDIA, Santa Clara

Oh D, Bakker T (1998) Utilization bounds for n-processor rate monotone scheduling with static processor assignment. Real-Time Syst 15:183–192. https://doi.org/10.1023/A:1008098013753

Olmedo IS, Capodieci N, Martinez JL, Marongiu A, Bertogna M (2020) Dissecting the CUDA scheduling hierarchy: a performance and predictability perspective. In: 2020 IEEE real-time and embedded technology and applications symposium (RTAS). IEEE. https://doi.org/10.1109/RTAS48715.2020.000-5

Othman H, Aji Y, Fakhreddin F, Al-Ali A (2006) Controller area networks: evolution and applications. In: 2006 2nd international conference on information & communication technologies, vol 2, pp 3088–3093. https://doi.org/10.1109/ICTTA.2006.1684909

Otterness N, Anderson JH (2020) AMD GPUs as an alternative to NVIDIA for supporting real-time workloads. In: Völp M (ed) 32nd Euromicro conference on real-time systems (ECRTS 2020), Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Leibniz International Proceedings in Informatics (LIPIcs), vol 165, pp 10:1–10:23. https://doi.org/10.4230/LIPIcs.ECRTS.2020.10

Ouyang T, Li R, Chen X, Zhou Z, Tang X (2019) Adaptive user-managed service placement for mobile edge computing: an online learning approach. In: IEEE INFOCOM 2019-IEEE conference on computer communications. IEEE, pp 1468–1476

Pang AC, Chung WH, Chiu TC, Zhang J (2017) Latency-driven cooperative task computing in multi-user fog-radio access networks. In: 2017 IEEE 37th international conference on distributed computing systems (ICDCS). IEEE, pp 615–624

Pattnaik A, Tang X, Jog A, Kayiran O, Mishra AK, Kandemir MT, Mutlu O, Das CR (2016) Scheduling techniques for GPU architectures with processing-in-memory capabilities. In: Proceedings of the 2016 international conference on parallel architectures and compilation, PACT'16. Association for Computing Machinery, New York, pp 31–44. https://doi.org/10.1145/2967938.2967940

Perkins C, Royer E (1999) Ad-hoc on-demand distance vector routing. In: Proceedings WMCSA'99. Second IEEE workshop on mobile computing systems and applications, pp 90–100. https://doi.org/10.1109/MCSA.1999.749281

RealTimeEngineers (2003) Freertos. https://www.freertos.org/. Accessed: 03 Aug 2023

Regnier P, Lima G, Massa E, Levin G, Brandt S (2011) Run: optimal multiprocessor real-time scheduling via reduction to uniprocessor. In: IEEE real-time systems symposium. IEEE, pp 104–115

Ren J, Yu G, Cai Y, He Y, Qu F (2017) Partial offloading for latency minimization in mobile-edge computing. In: GLOBECOM 2017-2017 IEEE global communications conference. IEEE, pp 1–6

Rossbach CJ, Currey J, Silberstein M, Ray B, Witchel E (2011) Ptask: operating system abstractions to manage GPUs as compute devices. In: Proceedings of the twenty-third ACM symposium on operating systems principles, pp 233–248

SAE (2021) Arinc653:avionics application software standard interface. https://www.sae.org/standards/content/arinc653p0-3/. Accessed: 03 Aug 2023

Safaric S, Malaric K (2006) Zigbee wireless standard. In: Proceedings ELMAR 2006, pp 259–262. https://doi.org/10.1109/ELMAR.2006.329562

Saifullah A, Xu Y, Lu C, Chen Y (2010) Real-time scheduling for wirelesshart networks. In: 2010 31st IEEE real-time systems symposium, pp 150–159. https://doi.org/10.1109/RTSS.2010.41

Samaddar A, Easwaran A, Tan R (2020) A schedule randomization policy to mitigate timing attacks in wirelesshart networks. Real-Time Syst 56:452–489

Silberschatz A, Gagne G, Galvin PB (2018) Operating system concepts, 10th edn. Wiley. https://www.os-book.com/OS10/

Sprunt B, Sha LR, Lehoczky JP (1989) Scheduling sporadic and aperiodic events in a hard real-time system. Final report. https://resources.sei.cmu.edu/asset_files/TechnicalReport/1989_005_001_15749.pdf

Sun C, She C, Yang C (2017) Energy-efficient resource allocation for ultra-reliable and low-latency communications. In: GLOBECOM 2017-2017 IEEE global communications conference. IEEE, pp 1–6

Takagi H, Kleinrock L (1984) Optimal transmission ranges for randomly distributed packet radio terminals. IEEE Trans Commun 32(3):246–257. https://doi.org/10.1109/TCOM.1984.1096061

Tan H, Han Z, Li XY, Lau FC (2017) Online job dispatching and scheduling in edge-clouds. In: IEEE INFOCOM 2017-IEEE conference on computer communications. IEEE, pp 1–9

Tanenbaum AS, Bos H (2022) Modern operating systems, 5th edn. Pearson, Boston

Tindell K, Burns A, Wellings A (2000) Calculating controller area network (CAN) message response times. Control Eng Pract 3:1163–1169. https://doi.org/10.1016/0967-0661(95)00112-8

Tong L, Li Y, Gao W (2016) A hierarchical edge cloud architecture for mobile computing. In: IEEE INFOCOM 2016-The 35th annual IEEE international conference on computer communications. IEEE, pp 1–9

Vu TT, Van Huynh N, Hoang DT, Nguyen DN, Dutkiewicz E (2018) Offloading energy efficiency with delay constraint for cooperative mobile edge computing networks. In: 2018 IEEE global communications conference (GLOBECOM). IEEE, pp 1–6

Vuduc R, Choi J (2013) A brief history and introduction to GPGPU. Springer, Boston, pp 9–23. https://doi.org/10.1007/978-1-4614-8745-6_2

Wei YH, Leng Q, Han S, Mok AK, Zhang W, Tomizuka M (2013) Rt-wifi: real-time high-speed communication protocol for wireless cyber-physical control applications. In: 2013 IEEE 34th real-time systems symposium, pp 140–149. https://doi.org/10.1109/RTSS.2013.22

WindRiverSystems (1987) Windriver vxworks. https://www.windriver.com/products/vxworks. Accessed: 03 Aug 2023

Xiao Y, Krunz M (2017) Qoe and power efficiency tradeoff for fog computing networks with fog node cooperation. In: IEEE INFOCOM 2017-IEEE conference on computer communications. IEEE, pp 1–9

Xu F, Xu J, Chen J, Chen L, Shang R, Zhou Z, Liu F (2023) igniter: interference-aware GPU resource provisioning for predictable DNN inference in the cloud. IEEE Trans Parallel Distrib Syst 34(3):812–827. https://doi.org/10.1109/TPDS.2022.3232715

Yandrofski T, Chen J, Otterness N, Anderson JH, Smith FD (2022) Making powerful enemies on NVIDIA GPUs. In: 2022 IEEE real-time systems symposium (RTSS). IEEE, pp 383–395. https://doi.org/10.1109/RTSS55097.2022.00040

Yao J, Lu Q, Tian R, Li K, Guan H (2023) An economy-oriented GPU virtualization with dynamic and adaptive oversubscription. IEEE Trans Comput 72(5):1371–1383. https://doi.org/10.1109/TC.2022.3199998

Yaqub U, Sorour S (2018) Multi-objective resource optimization for hierarchical mobile edge computing. In: 2018 IEEE global communications conference (GLOBECOM). IEEE, pp 1–6

Yu K, Gidlund M, Åkerbergy J, Björkman M (2013) Low jitter scheduling for industrial wireless sensor and actuator networks. In: IECON 2013 – 39th annual conference of the IEEE industrial electronics society, pp 5594–5599. https://doi.org/10.1109/IECON.2013.6700050

Yu R, Xue G, Zhang X (2018) Application provisioning in fog computing-enabled internet-of-things: a network perspective. In: IEEE INFOCOM 2018-IEEE conference on computer communications. IEEE, pp 783–791

Zeng D, Gu L, Guo S, Cheng Z, Yu S (2016) Joint optimization of task scheduling and image placement in fog computing supported software-defined embedded system. IEEE Trans Comput 65(12):3702–3712

Zhang DY, Wang D (2019) An integrated top-down and bottom-up task allocation approach in social sensing based edge computing systems. In: IEEE INFOCOM 2019-IEEE conference on computer communications. IEEE, pp 766–774

Zhang L, Geng S (1998) The complexity of the 0/1 multi-knapsack problem. J Comput Sci Technol 1:46–50. https://doi.org/10.1007/BF02943300

Zhao H, Cui W, Chen Q, Guo M (2023) Ispa: exploiting intra-SM parallelism in GPUs via fine-grained resource management. IEEE Trans Comput 72(5):1473–1487. https://doi.org/10.1109/TC.2022.3214088

Zheng X, Cai Z, Li J, Gao H (2017) A study on application-aware scheduling in wireless networks. IEEE Trans Mob Comput 16(7):1787–1801. https://doi.org/10.1109/TMC.2016.2613529

Zou A, Li J, Gill CD, Zhang X (2023) RTGPU: real-time GPU scheduling of hard deadline parallel tasks with fine-grain utilization. IEEE Trans Parallel Distrib Syst 34(5):1450–1465. https://doi.org/10.1109/TPDS.2023.3235439

Secure Processor Architectures

5

Nikhilesh Singh, Vinod Ganesan, and Chester Rebeiro

Contents

Introduction 172
Modern CPU Microarchitecture 173
Micro-architectural Attacks 176
Transient Micro-architectural Attacks 179
Meltdown and Spectre-Like Attacks 181
Micro-architectural Data Sampling Attacks 184
Countermeasures 189
Prevention-Based Countermeasures 189
Detection-Based Countermeasures 194
Conclusions 194
References 195

Abstract

In the last two decades, the evolving cyber-threat landscape has brought to center stage the contentious trade-offs between security and performance of modern microprocessors. The guarantees provided by the hardware to ensure no violation of process boundaries have been shown to be breached in several real-world scenarios. While modern CPU features such as superscalar, out-of-order, simultaneous multi-threading, and speculative execution play a critical role in boosting system performance, they are central for a potent class of security attacks termed transient micro-architectural attacks. These attacks leverage shared hardware resources in the CPU that are used during speculative and out-of-order execution to steal sensitive information. Researchers have used these

N. Singh · V. Ganesan · C. Rebeiro (✉)
Indian Institute of Technology Madras, Chennai, India
e-mail: nik@cse.iitm.ac.in; vinodg@cse.iitm.ac.in; chester@cse.iitm.ac.in

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_10

attacks to read data from the operating systems (OS) and trusted execution environments (TEE) and to even break hardware-enforced isolation.

Over the years, several variants of transient micro-architectural attacks have been developed. While each variant differs in the shared hardware resource used, the underlying attack follows a similar strategy. This chapter presents a panoramic view of security concerns in modern CPUs, focusing on the mechanisms of these attacks and providing a classification of the variants. Further, the authors discuss state-of-the-art defense mechanisms towards mitigating these attacks.

Keywords

Microprocessors · Architecture · Security · Leakages · Secure processor design · Micro-architectural attacks · Transient execution attacks · Hardware countermeasures · Software countermeasures · Micro-architectural attack prevention · Micro-architectural attack detection

Introduction

For over half a century, microprocessor research has focused on improving performance. Various micro-architectural features such as cache memories, branch prediction, superscalar, speculative, and out-of-order execution were developed to facilitate this. While some of these features, for example, the cache memory, were introduced to hide the latency of slow components, others like branch predictors helped hide overheads due to operations that slow down program execution. Features like out-of-order execution and speculative execution were introduced to better utilize available resources. Side-by-side features were incorporated in processors to support better multi-programming. Features such as multi-core processors and hardware multi-threading were incorporated to allow multiple users to simultaneously share a processor. These features accelerated new computing paradigms, especially cloud computing, where multiple users simultaneously share common hardware, thereby drastically reducing computation costs.

A critical aspect of the cloud computing paradigm is the isolation between users. To isolate one user's program from another, security schemes such as protection rings, segmentation, page table access controls bits, virtualization support, hardware-based security, crypto-accelerators, and trusted execution environments were introduced. Very soon, it was realized that these security schemes were insufficient. The shared hardware became a source of information leaks that could undermine the isolation provided by the processor. These attacks, popularly known as *micro-architectural attacks*, made use of shared hardware resources to glean sensitive information such as cryptographic keys, web pages visited, user passwords, and keystrokes. Different strategies such as time-driven attacks, Prime+Probe, Flush+Reload, and Evict+Time were proposed for this purpose. In a

cloud computing environment, these attacks could leak information from one user to another, in spite of having all security features enabled.

In 2018, two potent micro-architectural attack variants were proposed, namely, Meltdown (Lipp et al. 2018) and Spectre (Kocher et al. 2019), that exploited the speculative and out-of-order execution features present in microprocessors. These attacks leveraged the fact that a processor's speculation may not always be correct. When speculation goes wrong, the speculatively executed instructions, called transient instructions, need to be discarded, and the CPU should be rolled back to a previous state. However, this rollback is not always perfect. The CPU would still have a reminisce of the transient instructions. Researchers showed how this reminisce can be used to leak secrets. These attacks, which came to be called *transient micro-architectural attacks*, could read the contents of any memory region, including the OS memory. It could also read memory from trusted enclaves, even though the enclaves used encrypted memory.

Since 2018, there have been several variants of transient micro-architectural attacks including Zombieload (Schwarz et al. 2019a), Foreshadow (Bulck et al. 2018), Rogue In-Flight Data Load (RIDL) (van Schaik et al. 2019), Fallout (Canella et al. 2019), Load Value Injection (LVI) (Bulck et al. 2020), and Crosstalk (Ragab et al. 2021). Each variant found a new vulnerability that could bypass isolation in the CPU. Many of these attacks are not easily prevented by software patches. For those that can, the patches have huge performance penalties. It would require fundamental changes in the CPU design to mitigate these attacks in hardware.

This chapter would provide an introduction to transient micro-architectural attacks. Starting from Meltdown and Spectre, the authors would dwell on the basic principle of the attacks. This would be useful in distinguishing between the various attack classes and discussing the available mitigation techniques. Section "Modern CPU Microarchitecture" provides a background of modern CPU micro-architecture and also gives an introduction to micro-architectural attacks. Section "Transient Micro-architectural Attacks" discusses transient micro-architectural attacks and classifies them. Section "Countermeasures" discusses the defenses for these attacks, while the final section has the concluding remarks.

Modern CPU Microarchitecture

Notions of Security in Microprocessors. Beyond functional correctness, modern microprocessors attempt to enforce a root of trust to mitigate the ever-growing array of attacks. The goal of such approaches is to enable secure booting and provide platform to launch trusted execution environments (TEE) post boot-up. These TEEs, such as ARM Trustzone and Intel Software Guard Extensions (SGX) (Schunter 2016), ensure that the process boundaries guaranteed by the hardware are not violated by other processes. For example, the Intel SGX adopted in 2015 is a TEE feature supported by commercial processors that provide private regions of memory for programs. These regions are known as enclaves and cannot be accessed even from privileged software like the operating system. This is achieved by encrypting

enclave code and data present in the DRAM. Decryption is done when the code or data is fetched into the processor. Thus, the contents of an enclave, when in RAM, are always in an encrypted form and not accessible to any code outside the enclave regardless of the privilege levels. In recent years, however, researchers have shown that such trusted execution environments are not a panacea against the threat of transient micro-architectural attacks (Bulck et al. 2018; Weisse et al. 2018). The potency of these attacks is one of the reasons that led to the deprecation of Intel SGX from upcoming desktop processors (Intel Corporation 2021, 2022), posing further open questions regarding the security of hardware designs. In this section, we explore the premise of such attacks on the micro-architecture from first principles, starting with a background on the working of transient instruction in superscalar CPUs.

Transient Instructions in Superscalar CPUs. Figure 1 shows a block diagram of a superscalar CPU. In every clock cycle, multiple instructions are fetched from the instruction cache into an instruction/decode buffer which forms the frontend. The instructions are decoded into a set of micro-ops and are continuously fed to the execution engines, such as Arithmetic and Logic Units (ALU) and Floating Point Units (FPU), through a dispatcher of allocation queues. The scheduler ensures that the issue is possible only if the functional unit is available and the operands used by the instruction are up to date. The instructions to the functional units can be issued out of order, and based on a speculation, for example, the CPU can predict the outcome of a branch and speculatively execute instructions at the predicted branch target. The results from these speculatively executed instructions are stored in a temporary buffer and committed to registers and memory only when the speculation turns out correct. On the other hand, if the speculation is wrong, for example, a branch is mispredicted, the results from the speculatively executed instructions are dropped and not committed. These instructions are known as *transient instructions*. Besides branch mispredictions there are several reasons that can cause transient instructions. For instance, a userspace program executing a load or store instruction from an illegal memory, for example, from the kernel space, can result in a memory exception and also transient instructions. Another instance is of bound check instructions that identify if an index is within an array bound. Memory operations following the bounds can be speculatively executed with any arbitrary out-of-bound index.

In addition to the out-of-order and speculative execution of processes, many modern CPUs support the execution of multiple programs simultaneously. This feature is known as symmetrical multi-threading (SMT). Instructions from two or more programs simultaneously execute in a single pipeline sharing hardware resources such as cache memories, branch prediction units, and various other on-chip resources.

Micro-architectural State. As instructions flow through the CPU, various registers, buffers, caches, and other memory structures in the CPU core store temporary data and results from the execution. While a few of these memory structures, for

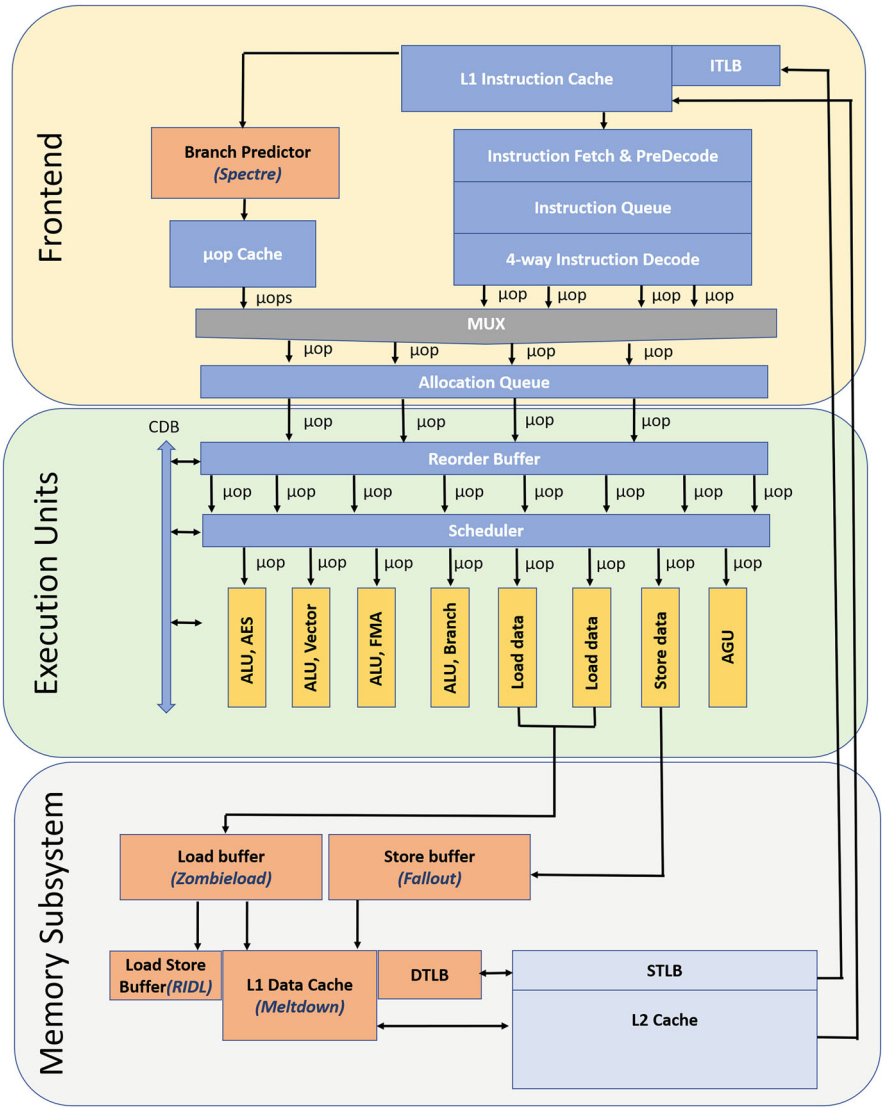


Fig. 1 An out-of-order superscalar processor with vulnerable components shaded in orange

instance, the general purpose registers, can be read or modified using instructions in the ISA, a significant portion of the structures are hidden and inaccessible from software. To enforce separation between applications, system software ensures that the data present in the ISA visible shared memory structures of one application cannot be read or modified by another application. For example, during a context switch, general purpose registers are either invalidated or loaded with the context of

the next process that executes, thus achieving a temporal separation between the two processes. In multi-core or multi-threaded CPUs on the other hand, the ISA visible memory structures are duplicated enforcing spatial separation.

Unlike the visible structures, the hidden memory structures in the CPU, such as cache memories and branch prediction units, are not always spatially and temporally separated between applications. They retain their values across context switches and are possibly shared in multi-core and multi-threaded CPUs. For example, a cache line that holds data from one application can be evicted by another application. Similarly, a branch predictor trained on branches in one application can influence the outcome of a prediction in another application. At first glance, this may seem innocuous as the structures are hidden from software. However, researchers have found that one application can indirectly affect another by these shared hidden memory structures. This has led to a series of security vulnerabilities, commonly grouped in a category called micro-architectural attacks. The red regions in Fig. 1 are modules in the processor with demonstrated security vulnerabilities. Researchers have used these vulnerabilities to break cryptographic algorithms, read operating system data, and break trusted execution environments.

Researchers have used these vulnerabilities to break cryptographic algorithms (Bernstein 2005; Percival 2005), design keyloggers (Ristenpart et al. 2009), fingerprint websites (Shusterman et al. 2019), break security features like Address Space Layout Randomization (Barresi et al. 2015; Gras et al. 2017; Hund et al. 2013), and leak sensitive information from the operating system (Kocher et al. 2019; Lipp et al. 2018) and trusted enclaves (Bulck et al. 2018; Weisse et al. 2018). They have been applied on a variety of devices ranging from mobile phones to cloud computing servers. The next section provides a brief introduction to micro-architectural attacks.

Micro-architectural Attacks

This section introduces micro-architectural attacks using the example of cache memories. The cache memory is a high-speed memory placed between the CPU and RAM to cache recently used instructions and data. It can be simultaneously shared by multiple applications in a CPU core. Due to its small size, it can be the cause of contention when applications compete for the same cache line. The authors explain the fundamental working of micro-architectural attacks by using three examples. The first uses a prime and probe algorithm on a shared cache memory, while the second is an algorithm called flush and reload that uses shared library code. The third is an evict and time algorithm on the cache memory.

Prime+Probe Attacks. Prime and probe forms for the basis for several micro-architectural attacks. It exploits the variance in the execution time caused by two applications that contend for the same shared hardware resource. The attack is discussed by showing an example of how the cache memory can be used to create

a covert communication channel between a high-privileged application and a low-privileged application. Similar channels have been used on other shared resources as well, such as TLBs, branch prediction units, load and store buffers, and even DRAM.

Consider that the high-privileged application called the *sender* and the low-privileged application called the *receiver* share a common cache memory. For example, the sender and receiver execute simultaneously on a CPU with a shared L1 data cache memory. The objective of the covert channel is to use the increase in execution time due to contention in the cache memory to transmit a message from the sender to the receiver. A priori, the sender and receiver agree upon two cache sets $C0$ and $C1$ for communicating bit 0 and 1, respectively. The communication works as follows. **(1)** The receiver first performs memory load operations that fill both cache sets. This is called the *prime* phase and is done by loading data from addresses that map to sets $C0$ and $C1$ as shown in Fig. 2a. **(2)** Depending on the

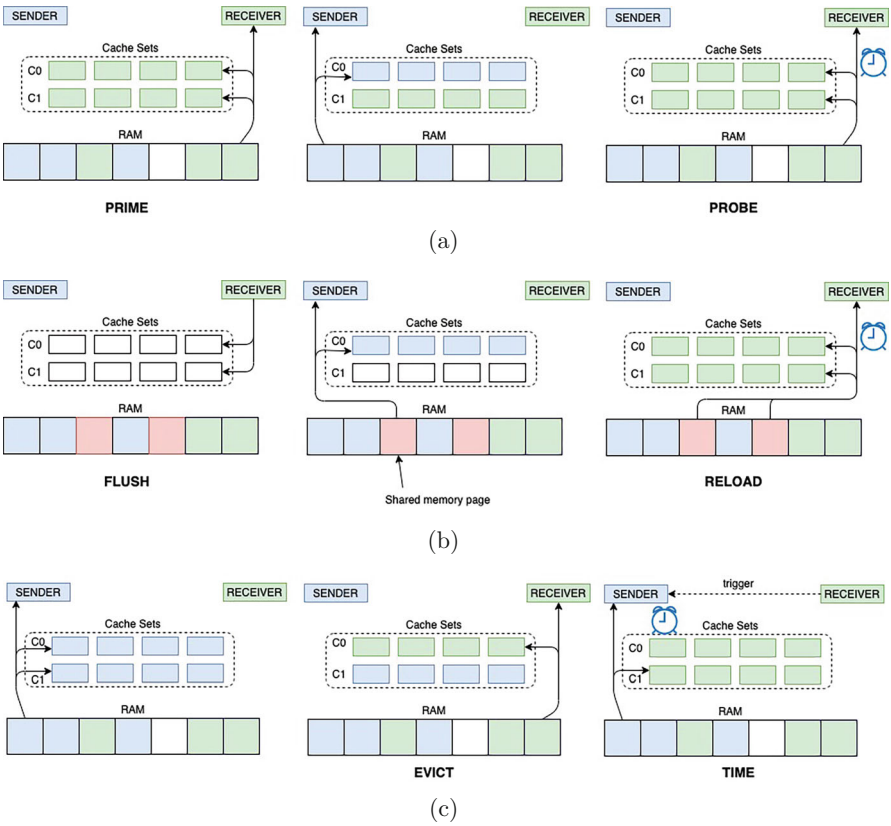


Fig. 2 Prime+Probe, Flush+Reload, and Evict+Time are the most common algorithms used to exfiltrate data in a micro-architectural attack. This figure demonstrates these algorithms in a covert channel that uses cache memory to transmit one bit of information from a sender to a receiver. (**a**) Prime+Probe. (**b**) Flush+Reload. (**c**) Evict+Time

message bit, the sender performs a memory operation to evict the receiver's data from the corresponding cache set. For example, to transmit a 0, the sender would evict the receiver's data from the cache set $C0$. **(3)** In the *probe* phase, the receiver repeats the memory operations in step (1), but this time also measures the execution time. Based on the execution time, the receiver can infer the transmitted bit since the memory access to the evicted cache set would take longer owing to the cache miss.

Prime+probe in micro-architectural attacks work similarly, except for the fact that the sender and receiver do not collude. Instead, the receiver primes sufficient number of sets in the cache (step (1)), waits for the sender to execute and evict one or more cache lines in these sets, and then performs a probe similar to step (3) to identify patterns in the sender's execution.

Flush+Reload Attacks. Unlike Prime+Probe attacks, where the information leakage is due to conflicts in the cache memory, in the Flush+Reload attacks, information leakage is caused without forcing cache conflicts. Consider, for instance, the high- and low-privileged applications sharing memory pages. Such sharing is common in systems that use shared libraries. A single copy of the shared library present in RAM is used by multiple applications. The time required to load data in a shared memory page depends on whether the data is in the cache or not. If present in the cache, the load will be considerably faster than if it is not present in the cache.

Consider that the sender and receiver of a covert channel decide on two shared regions of code or data, for example, in a shared library. These regions are chosen so that they map to distinct cache sets: $C0$ and $C1$. In step **(1)**, the receiver ensures that the data in these two regions are not in the cache shown in Fig. 2b. This is performed by a flush operation that evicts the addresses from the cache and is called the *flush* phase. On Intel x86 platforms, an instruction called `clflush` is used to perform this. The `clflush` takes an address as argument and flushes the addresses from all caches in the CPU. **(2)** In the second step, the sender performs a load operation to either $C0$ or $S1$ depending on whether it wants to transmit a 0 or 1, respectively. It would cause the data from one of the two shared regions to be fetched into the cache. **(3)** The receiver then performs loads on both addresses and measures the time taken. This is called the *reload* phase. Only one of these two loads would result in a cache hit. The time when there is a cache hit would be much shorter than the time when there is a cache miss. This difference in time can be used to infer the bit transmitted. Unlike the Prime+Probe attack techniques, Flush+Reload is independent of the cache attributes, like its associativity. It thus results in more portable attacks.

In transient micro-architectural attacks, the attacker defines an array. Similar to the covert channel, in step (1) the attacker ensures that no elements of the array are present in the cache memory. In step (2), the attacker triggers a transient load operation that forces exactly one element from the array to be loaded into cache. Similar to the step (3) in the covert channel, the attacker would do a reload to identify which element was loaded. In element of the array that is loaded transiently often reveal secret information, the operating system data.

Evict+Time Attacks. Evict+Time attacks closely resemble the Prime+Probe attacks. The difference is that the adversary is able to accurately measure the execution time of the sender application. While this is a strong assumption, there are certain scenarios where such measurements are possible, for example, when the adversary can trigger the execution of the sender and an observable event marks the end of its execution. In such cases, the duration between the trigger and the event serves as a measure of the execution time of the sender.

Consider again the covert channel between the high-privileged sender and low-privileged receiver application. The assumption at the start is that both cache sets, $C0$ and $C1$, have the sender's data. In the second step, the receiver evicts one of the cache sets, say $C0$ as shown in Fig. 2c. In the third step, it triggers the sender to execute and monitors the execution time of the sender. The sender would transmit a 0 or 1 by loading data from memory that maps to the $C0$ and $C1$ cache set, respectively. The time taken to perform this load differs for the 0 and 1 bit transmissions. Transmitting 0 will result in a cache miss; thus experiencing a longer execution time compared to transmitting 1. This difference in time is observed by the receiver to infer the transmitted bit.

Transient Micro-architectural Attacks

When transient instructions execute, the hidden states of the CPU are modified. While the results of a transient operation are discarded after the speculation is proved wrong, the hidden state of the CPU is not rolled back. Thus, transient instructions have a permanent impact on the CPU state. Consider, for example, the following code snippet.

```
I1. cmp r0, r1
I2. jne <dest_addr> /* branch to dest_addr, if r0 != r1 */
I3. mov r2, Addr1
I4. add r2, r1      /* r2 = r2 + r1 */
I5. load r3, r2     /* r3 = memory corresponding to (r2) */
```

In an out-of-order processor, instructions `I3`, `I4`, and `I5` can be transiently executed if the CPU mispredicts the branch outcome at `I2`. If the memory load in `I5` results in a cache miss, it causes data at the address present in `r2` to be loaded into cache. Due to the misprediction, the CPU would discard the results of instruction `I3`, `I4`, and `I5`; however, it would not roll back the state of the cache memory. Thus, data corresponding to the memory load in `I5` would persist in the cache even after the transient executions are dropped. In 2018, researchers showed that this reminiscence of a transient execution could lead to serious security vulnerabilities that could potentially compromise every application executing on the CPU. The two attacks, namely, Meltdown and Spectre, that were proposed in 2018 showed how this reminiscence could undermine the security of application software on a variety of commercial microprocessors. Since then, several variants of such transient attacks have been proposed. They form a new class of extremely powerful micro-

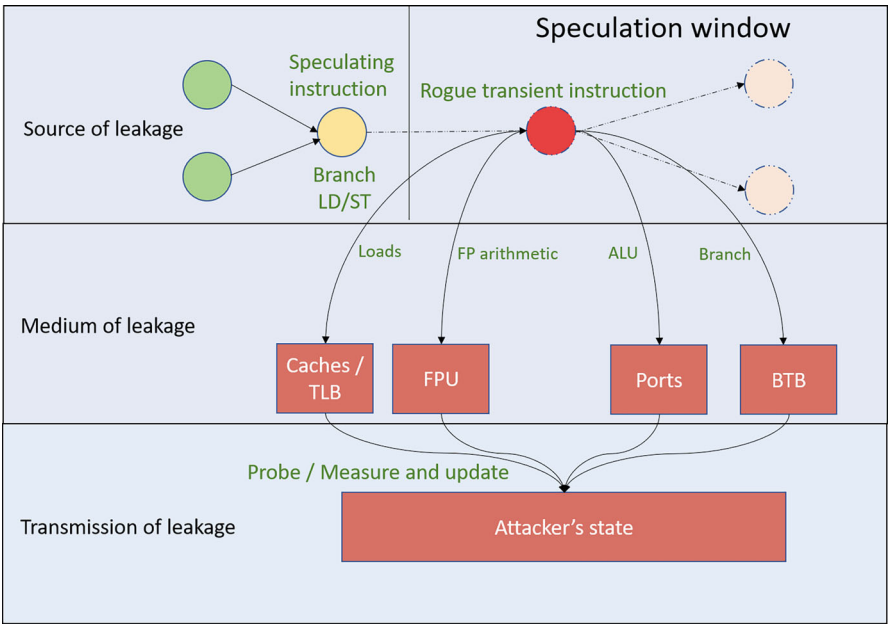


Fig. 3 In a transient attack, the transient instruction modifies the hidden states of CPU like cache memories, FPU, and ports, in a manner that depends on secret information. In the next stage, the attacker exfiltrates these secrets from the hidden states

architectural attacks and have come to be known as transient micro-architectural attacks or simply *transient attacks*.

A typical transient attack has three stages, as shown in Fig. 3. The first stage disrupts the flow of program execution by forcing an exception or by inducing a misprediction that could trigger transient execution. In the next stage, the attacker relies on one or more of the transiently executed instructions to modify a hidden CPU state, such as the cache memory, branch predictor, or an internal buffer. The transient instruction is designed in a way so that the modification in the hidden state is correlated with a secret. The secret, for instance, can be keys of cryptographic ciphers, kernel code or data regions, or any other sensitive information. Due to the exception or the misprediction that occurred in the second phase, the transiently executed instructions are discarded, while the hidden micro-architectural states remain unaltered. In the final stage, the attacker exfiltrates information from the hidden micro-architectural state using an algorithm like Prime+Probe, Evict+Time, or Flush+Reload, to glean information about the secret.

After the initial attacks, vis-à-vis Meltdown and Spectre, several variants of transient attacks have appeared in the literature (Bhattacharyya et al. 2019; Bulck et al. 2018, 2020; Canella et al. 2019; Chen et al. 2020; Ragab et al. 2021; Schwarz et al. 2019a,b; van Schaik et al. 2019; Weisse et al. 2018). Each new attack identified a new medium of leakage. Broadly, these attacks can be categorized into two

Table 1 Transitive micro-architectural attacks

Attack	Requirement	Source of leakage
–Meltdown and Spectre-like attacks–		
Meltdown (Lipp et al. 2018)	SMT	Transitive load
Spectre (Kocher et al. 2019)		BPU
Foreshadow (Bulck et al. 2018)		Transitive load
–Micro-architectural data sampling–		
RIDL (van Schaik et al. 2019)	SMT	Line feed buffer
Fallout (Canella et al. 2019)		Store buffer
Zombieload (Schwarz et al. 2019a)		Line feed buffer
LVI (Bulck et al. 2020)		Store buffer
Crosstalk (Ragab et al. 2021)		Staging buffer

classes based on the micro-architectural medium used for the leakage. The first is address-controllable transient attacks like Meltdown and Spectre, while the others are based on micro-architectural data sampling from internal buffers. While at a high level, the stages in both categories are the same and follow Fig. 3, there are subtle differences between the two classes. Address-dependent attacks like Meltdown and Spectre use micro-architectural components like cache memories or branch prediction units as a medium for leakage. In these attacks, data (or instructions) placed in strategic memory addresses are transiently loaded (or executed). For example, in the covert channels described in section "Micro-architectural Attacks", an address is used to select a cache set. The choice of the cache set is used as a medium for information leakage. In micro-architectural data sampling attacks like Zombieload and Crosstalk, on the other hand, it is not the address that is critical. Instructions are crafted so as to snoop into internal buffers like re-order buffers, line-fill buffers, and load and store buffers. Table 1 classify the known attacks into these two categories.

Meltdown and Spectre-Like Attacks

These attacks require the knowledge memory regions of interest, and the attacker can target them specifically. Attacks like Meltdown (Lipp et al. 2018), Spectre (Kocher et al. 2019), and Foreshadow (Bulck et al. 2018) fall in to this category. The upcoming sections look into each of these attacks to elaborate on their design and mechanisms.

Meltdown. CPUs use protection rings to isolate privileged code. For example, Intel CPUs have four rings: Ring 0 to Ring 3. Privileged code, such as the operating system's kernel, is assigned to Ring 0, while user processes are assigned to Ring 3. The hardware ensures that during regular operations, code executing in Ring 3 cannot read or write to Ring 0, thus isolating the kernel's code and data from

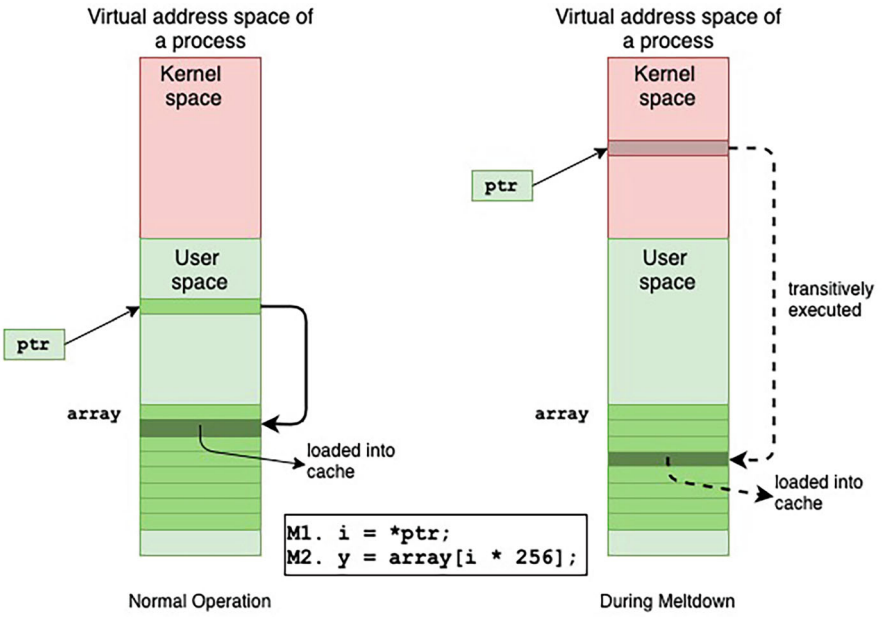


Fig. 4 Transitive execution of a memory load instruction causes data from `array` to be loaded into cache memory. Unlike the visible micro-architectural state, the cache contents are not rolled back when transient instructions are discarded. The contents of the cache can be gleaned using techniques such as Prime+Probe or Flush+Reload

userspace programs. The Meltdown attack exploits transient execution to read kernel data from a user program, thus breaching the isolation provided by the protection rings.

Prior to 2018, the kernel was mapped into the virtual address space of every process, as shown in Fig. 4. This simplifies system calls and interrupt handling. Since the kernel was in Ring 0, a user function would not be able to directly access the kernel. The Meltdown attack showed how a userspace transient memory load or store operation to a kernel address caused the data to be loaded into the cache memory. This data could then be gleaned using one of the micro-architectural algorithms like Prime+Probe or Flush+Reload (section "Micro-architectural Attacks").

In the first stage of Meltdown, the attacker writes code (Institute of Applied Information Processing Communications) as shown in Fig. 4 that would perform a load from a kernel address. Specifically `ptr` is made to hold a kernel address. In the ideal case, this should have immediately created an exception because a user instruction is trying to read kernel data. However, modern CPUs are designed in a way that delays the exception, allowing subsequent instructions to be transiently executed. The contents of the kernel space data would thus be loaded into the register `i`, which is then used to load an element from the array into `y`. During this process, `y` is also stored in the cache memory. Notice that the array is indexed based on

the kernel data. All of these instructions are transiently executed. At the time of throwing the exception, the CPU would discard the new values of `i` and `y`, but will not roll back the cache memory.

In the final stage of Meltdown, either the Flush+Reload or the Prime+Probe can be used to identify the cache set that holds the loaded `array` data, thus revealing information about the kernel data. With the Flush+Reload, for instance, the attacker would first ensure that all `array` elements are flushed from the cache before the transient instructions `M1` and `M2` execute. Post their execution, exactly one element corresponding to `y` would be present in the cache. The cache set that holds `y` can be inferred by measuring execution time to load each `array` element. The cache set containing `y` would have the shortest load time due to a cache hit. All other elements, by virtue of the initial flush, would result in cache misses.

Spectre. While the Meltdown attack makes use of an illegal load or store memory operation to induce a transient execution, Spectre makes use of mispredicted branches. Modern microprocessors have a Branch Prediction Unit (BPU) that speculates the direction and the target address of a branch during program execution. The prediction is done by learning patterns in taken and not-taken branches from the branch history. For example, consider the following code snippet, where `array1_size` is the size of `array1` and is used to check the bounds of the index `x`. Statements `S2` and `S3` are executed only if `x` is within bounds.

```
S1. if (x < array1_size){
S2.     i = array1[x];
S3.     y = array2[i * 256];
S4. }
```

If the snippet is executed repeatedly with legal values of `x`, the BPU would learn the execution pattern and speculatively execute statements `S2` and `S3`. The results in `i` and `y`, however, would be committed only after the check `x < array1_size` is completed. After a while of such repeated executions, if `x` is made illegal (i.e., $x \geq$ `array1_size`), the BPU would predict incorrectly leading to transiently executed `S1` and `S2`. The two transient memory operations would load data into cache. The misprediction would ignore the new values computed for `i` and `y` but not rollback the cache memory. The final stage of Spectre is similar to Meltdown and uses micro-architectural attack techniques like Evict+Time and Flush+Reload to glean information about `array1[x]` from the cache memory. For example, if `array1[x]` corresponds to a kernel region, the attack would reveal the contents of the kernel location.

Spectre is one of the most powerful of all transient attacks because it is very difficult to mitigate. Over the years, multiple variants of Spectre have been proposed that exploit the different components of branch speculation in the processor. The different variants of Spectre attempt to tune different tables in the BPU. For example, Kocher et al. (2019) and Schwarz et al. (2019b) exploits the Path History Table (PHT), while Bhattacharyya et al. (2019), Chen et al. (2020), and Kocher et al. (2019) exploits the Branch Target Buffer (BTB), and Koruyeh et al. (2018) and Maisuradze and Rossow (2018) use the Return Stack Buffers (RSB).

Foreshadow. The Meltdown attack breaches the isolation provided by CPU's protection rings, thereby reading kernel data from a user program. Since 2015, Intel has added another level of protection in its processors. The Intel Security Guard Extension (SGX) is a feature supported by commercial processor variants (deprecated, 11th generation Intel Core onwards (Intel Corporation 2021, 2022)) that provide private regions of memory, called enclaves, for programs. It is ensured that the contents of an enclave, when in RAM, are always in an encrypted form, barring any access to a piece of code outside the enclave, regardless of the privilege levels.

The Foreshadow attack makes use of the fact that data in the SGX enclaves are stored in the plain form in the L1 cache. This allows transient instructions to compute on the cached secrets. The challenge is to cache secret data in the enclave and use them in transient operations. Given this, the Foreshadow works very similar to Meltdown (Lipp et al. 2018). It uses a local buffer that is transiently accessed at indices that depend on secret data stored in the enclave. Now that the entries from the buffer are in the cache, the attacker simply deploys the Flush+Reload technique to establish the secret from the enclave.

In principle, Foreshadow attacks are a variant of the Meltdown attack that use the same vulnerability, not just to read kernel memory from user space, but to rupture security mechanisms of Intel SGX (Schunter 2016) that attempt to provide secure enclave protection domains. An improvement on the basic attack is Foreshadow-NG (Next Generation) (Weisse et al. 2018) which has the potential to read any information that comes to the L1 cache affecting Virtual Machines (VMs), hypervisors (VMM), operating system (OS) kernel memory, and System Management Mode (SMM) memory.

Micro-architectural Data Sampling Attacks

Supporting speculative and out-of-order execution in a microprocessor often requires buffers at several locations in the CPU pipeline that temporary stores details about in-flight instructions. For example, Reorder Buffers (ROBs) are used to track instructions executed out of order and commit their results in the correct order. Other examples are the store buffers, used to track pending stores involved in optimizations. Micro-architectural Data Sampling (MDS) attacks are able to snoop into these temporary buffers to glean secret data from other applications. Unlike Meltdown and Spectre like attacks, MDS attacks are not tied to specific memory addresses, making it almost impossible to mitigate from software. This section summarizes the known MDS attacks.

Rogue In-Flight Data Load (RIDL). In traditional cache memories, a cache miss would block any further memory requests until the cache miss is serviced. In out-of-order CPUs, addresses corresponding to cache misses are stored in a line-fill buffer (LFB), so that subsequent memory requests can be serviced. This helps create a non-blocking cache. On receiving a memory request that results in a cache miss,

an entry in the LFB is created to store the requested address. Subsequently, when the memory block is fetched, it is stored in the LFB entry corresponding to the memory address. The block is also stored in the cache memory and forwarded to the CPU core. The RIDL attack is able to snoop into the line-fill buffer (LFB) to retrieve the data from the stored block. Interestingly, the attack does not depend on the address of the memory request, but only requires a cache miss that makes an entry in the LFB.

RIDL assumes that the attacker and victim share a common L1 cache memory. The steps of the attack are shown in Fig. 5. The attacker first ensures that `buffer` is flushed from cache and then triggers the victim to execute a load instruction, say at address `A`. If this victim's load results in a miss in the L1 cache, a new entry would be created in the LFB which would store the physical address of `A`. The attacker, running on a different thread in the same core, issues a load to an address present in a new invalid page. Since this page is new, it would result in a TLB miss and

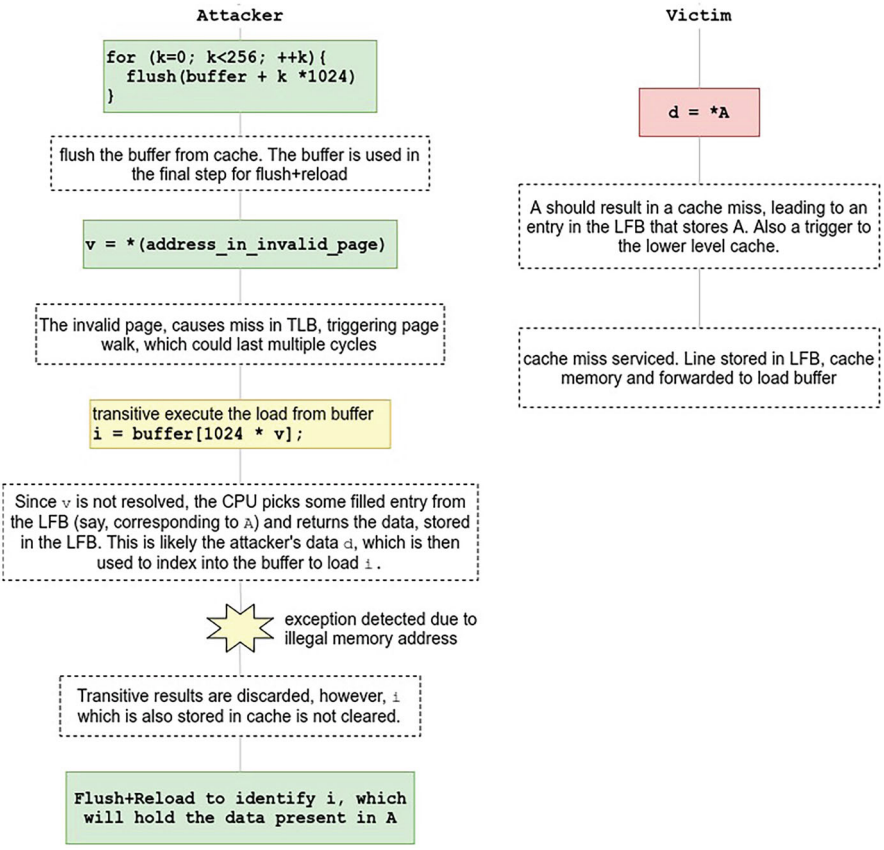


Fig. 5 In the RIDL attack, the attacker (in green) snoops into the line-fill buffer (LFB) to read the victim's sensitive data present in the address (A)

trigger a page table walk. The CPU would eventually detect that the load request is from an invalid page and mark it for exception. The exception is however thrown much later when the operation's results are committed in order. During this time, the memory load operation from `buffer[1024 * v]` would continue transitively using an arbitrary value of `v` picked from an entry in the LFB. The address parts of the LFB entry are not matched; therefore, with significant probability, the entry would correspond to the victim's load request at `A`, resulting in `v` holding the value of the victim's data `d`. Thus `buffer[1024 * v]` is indexed at a location that is dependent on `d`. The result is stored in `i`, as well as in a cache set. After the exception is thrown due to the illegal address, the transitive results in `v` and `i` are discarded; however, the cache is not rolled back. Flush+Reload is then used to identify `i`, thus revealing information about the attacker's data.

Zombieload. This attack (Institute of Applied Information Processing Communications) exploits LFB like RIDL, some unknown micro-architectural components, and the concept of microcode assists to mount the attack. Recall that an LFB tracks all load values that are not present in the L1 data cache and needs servicing from higher-level cache hierarchies. Whenever there are complex micro-architectural conditions, such as page faults, it can be handled in one of two ways: (i) the fault can be delegated to a software service routine, or (ii) one can employ *microcode assists*, where the fault is handled through a set of microcode routines, which is faster than delegating to a software. A microcode assist always triggers a pipeline flush resetting the architectural state. However, in-flight instructions still finish execution only to be discarded later. Similarly, the outstanding LFB entries are not discarded. To not incur additional delays in completing the execution of in-flight instructions, the LFB is allowed to load *stale values* for previous load or store instructions, altering the micro-architectural state and potentially allowing the leakage of data. This data can be gleaned by the process of data sampling explained above.

Though this attack looks similar to RIDL, the key contrast of this work is that the above leakage occurs even if the authors systematically ensure using Intel TSX (Intel) that there is no entry filled in Line Buffer during a cache miss. Intel TSX is a set of hardware extensions that enable one program or a program thread to acquire a lock on certain memory locations in the memory which is prohibited from being updated or used by any other program until released. This enables concurrent programming as the updates in these locations are done atomically by one program or a thread at a time. Within a TSX window, and during certain situations, a miss in the L1-cache never creates a line-buffer entry. However, even without LFB, the leak happens, rather surprisingly at a much higher rate. This suggests that Zombieload is working not only because of LFB but also due to other unknown micro-architectural components, such as FPU register file and store buffer.

Fallout. Out-of-order processors hide the latency associated with store operations by using a store buffer. On encountering a store operation, an entry is created in the buffer to hold the virtual address, physical address, and the value to be stored in memory. After the entry is created, subsequent operations in the program can

speculatively execute permitting the stores to complete asynchronously. If one of the subsequent operations is a load, the data from the store buffer is forwarded. This is called *store-to-load* forwarding. Such *store-to-load* forwarding is possible in two conditions:

- **Condition 1.** If the complete address in the load matches the complete address of an entry in the store buffer, then the value in the entry can be directly used.
- **Condition 2.** If the virtual to physical address translation for the load fails, and a few least significant bits match with an entry in the store buffer, then the value in the entry can be speculatively used.

In their paper Canella et al. (2019), the authors show how both these conditions can lead to transient attacks. The attacks arise from the fact that store-to-load forwarding can happen across security domains. It only requires either of the two conditions to be met. For example, the value in the store buffer entry will be forwarded just by matching address bits in the store buffer entry and the load operation. The store could be from the kernel, while the load from a userspace program.

The second condition leads to an attack called *Data Bounce* that is used to identify if a virtual address is valid (i.e., mapped to a physical address). The pseudo-code is shown in Fig. 6a. This attack can be used to break Address Space Layout Randomization (ASLR) (PaX; Bhatkar et al. 2003; Xu et al. 2003). The first condition leads to a vulnerability called *Write Transient Forwarding (WTF)*. The vulnerability can be used to snoop into stores from another process. Figure 6 provides more details about these attacks.

Load Value Injection (LVI). In traditional out-of-order processors, a store to a memory location followed by a subsequent load instruction to the same memory location can be slow as it comprises two sequential instruction executions involving costly memory accesses. However, a widely used optimization to alleviate this, as explained above in fallout, is to perform *store-to-load* forwarding that will forward the contents of the producing store directly to the consuming load if both the entries are present in the load/store buffer. However, the effective addresses of the load/store instructions are not resolved until later, and hence they are speculated instead. Therefore, during speculation there is a possibility that a wrong store forwards a value to the load. LVI uses this key principle to poison the data that the victim operates on to leak information. This is illustrated using Fig. 7. In this example, the `untrusted_arg` is sent by the attacker which the victim stores within its buffer space (trusted memory), termed as the poisoning phase. Now, in case there is a page fault caused when dereferencing trusted_ptr, the `trusted_ptr` erroneously receives value from the `untrusted_ptr` due to store-to-load forwarding within the load store buffers under speculation. This poisoned data is now the index variable for an array whose values can now be leaked through standard cache-based attacks such as Flush + Reload. Generally, the attack contains three phases: (i) Micro-architectural poisoning where the attacker prepares the injection of a poison value by

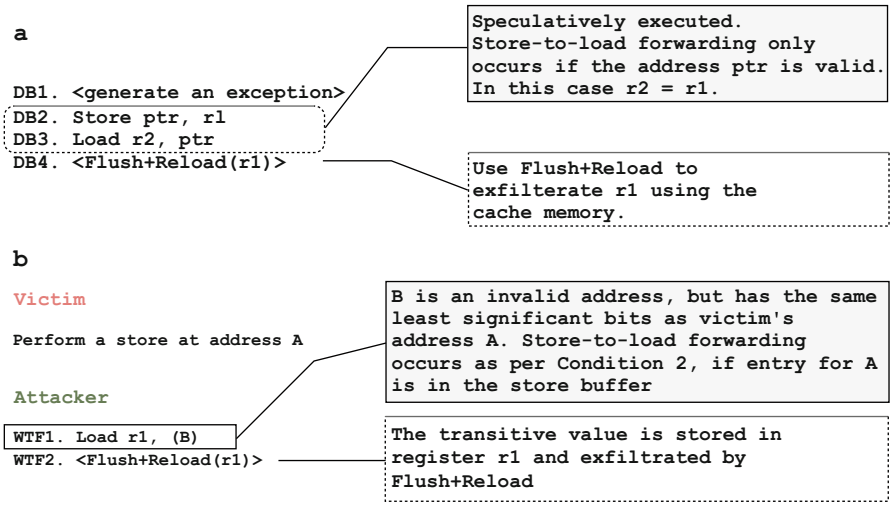


Fig. 6 Fallout makes use of store-to-load forwarding of data in the store buffer to a speculatively executed load operation. The load operation can be from a different security domain, for example, the kernel. The result of the load is stored in the `r1` register and exfilterated using Flush+Reload. Flush+Reload is similar way to previous attacks. The flush is done before the exception causing instruction, while the reload is done after the transitive execution is discarded. (**a**) Data Bounce occurs due to Condition 1. (**b**) Write transitive forwarding vulnerability occurs due to Condition 2

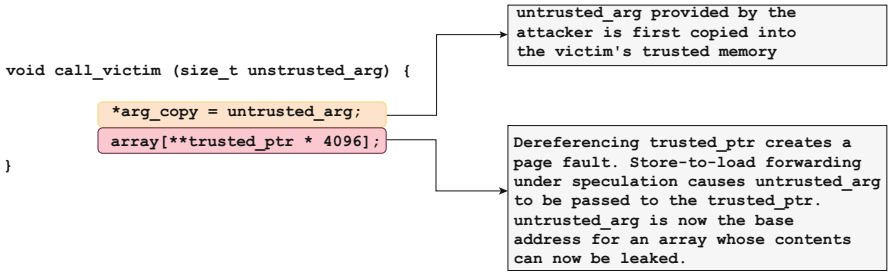


Fig. 7 In LVI, the attacker injects a malicious value through load forwarding and uses that to leak sensitive data

loading that in one of the micro-architectural buffers. (ii) The attacker then provokes the victim into executing instructions that cause a page fault or exception which triggers this store-to-load data poisoning. This can be done, for instance, by evicting a set of victim's virtual memory pages. (iii) Gadget-based secret transmission, where the attacker finds exploitable code gadgets that can leak data under incorrect transient execution behavior and lead the victim to that code gadget by carefully poisoning the data.

Crosstalk. Crosstalk demonstrates that MDS vulnerabilities exist beyond the CPU core through a shared memory buffer, called staging buffer, that is shared across multiple CPU cores. The authors identify several micro-instructions that touch the buffer. These instructions, if executed transiently, can potentially lead to leakage from one CPU core to another. One use case of Crosstalk is to leak hardware-generated random numbers that uses Intel's Secure Key Technology. The Secure Key technology makes use of an off-core hardware random number generator. The generator is initialized using the `RDSEED` instruction, and the random numbers are read using the `RDREAD` instruction. These form the basis of several cryptographic primitives including Intel's security enclaves. Executing either of these instructions touches the staging buffer. MDS attacks can be mounted on the buffer by transiently executing `RDRAND` and `RDSEED`, thus leaking the seed or the random numbers generated by the hardware random number generator.

Countermeasures

Since their discovery, there have been extensive efforts to design and develop countermeasures for transient micro-architectural attacks. The countermeasures can be broadly classified as prevention-based or detection-based. Prevention-based solutions attempt to stop the attack by thwarting the execution at one of the three phases (refer Fig. 3). Naïve preventive solutions, for instance, disable speculative execution, thus preventing any transient execution, the first stage of the attack. Another naïve preventive solution disables all timers, thus preventing timing channels. This would disable stage 3, i.e., the transmission of leakage. In contrast, detection-based solutions do not disable any feature; rather, they aim to identify patterns in the program execution that can be attributed to an attack. While preventive-based solutions have high overheads, detection-based solutions suffer from false positives. Over the last few years, there have been multiple detection-based and preventive-based solutions proposed. Table 2 provides a list of these solutions. This section provides a description and analysis of some of these existing solutions.

Prevention-Based Countermeasures

Figure 3 shows the stages of a transient attack. The attacker first identifies a source of leakage, as listed in Table 1. The next step involves the transient movement of data from the source to the medium of leakage. Finally, the attacker uses techniques established in section "Micro-architectural Attacks" to transfer the secret information from the medium. Thwarting any of these sequential stages is sufficient to prevent the attack. Different preventive countermeasures target attacks at different stages of their execution, as described in Table 2.

Prevention-based countermeasures provide a preemptive solution to these attacks. While the goal of all solutions is to disable potentially vulnerable behavior

Table 2 Countermeasures for transient micro-architectural attacks are classified as either prevention-based or detection-based. While prevention-based techniques aim to either modify or disable some functionality in the software or hardware, detection-based techniques rely on accurately identifying attacks from their run-time characteristics (*HW* hardware implementation, *SW* software implementation)

Stage of applicability	Paper	HW or SW?	Threat model	Reported overheads
–Prevention-based–				
Source of leakage	NDA (Weisse et al. 2019)	HW	Speculative execution attacks	4–32%
	Context (Schwarz et al. 2020)		Spectre-like	0–338%
	InvisiSpec (Yan et al. 2019)		Spectre-like	5–17%
	Safespec (Khasawneh et al. 2019)		Meltdown, Spectre-like	3%
	SpectreGuard (Fustos et al. 2019)		Spectre-like	8–20%
	Specshield (Barber et al. 2019)		Speculative execution attacks	21%
	Spectrum (González Abraham et al. 2018)		Spectre-like	2%
	MuonTrap (Ainsworth and Jones 2020)		Spectre-like	4%
	Invisible speculation (Sakalis et al. 2019)		Cache and memory side channels	11%
	Reversispec (Wu and Qian 2020)		Speculative load attacks	8.3%

Medium of leakage	Random-fill (Liu and Lee 2014)	HW	Contention- and reuse-based attacks	Negligible
	Newcache (Liu et al. 2016)		Contention- and reuse-based attacks	Negligible
	CEASER (Qureshi 2018)		Contention-based attacks	1%
	Encrypted-address cache (Qureshi 2019)		Contention-based attacks	1%
	Scattercache (Werner et al. 2019)		Cache leakage techniques (section "Micro-architectural Attacks")	2–4%
	DAWG (Kiriansky et al. 2018)		Cache timing attacks	4–7%
	SecDCP (Wang et al. 2016)		Timing side channels	12.5%
	MI6 (Bourgeat et al. 2019)		Spectre-like	16.4%
Transmission of leakage	Timewarp (Martin et al. 2012)	HW	Timing side channels	Negligible
	InvarSpec (Zhao et al. 2020)	SW	Speculative execution attacks	(Yan et al. 2019): 10.9%
	oo7 (Wang et al. 2018)	SW	Spectre-like	5.9%
	SPECCFI (Koruyeh et al. 2020)	SW	Spectre-like	1.9%
–Detection-based–				
Transmission of leakage	Cyclone (Harris et al. 2019)	SW	Cache leakage techniques	3.6%
	(Chiappetta et al. 2016)		Cache leakage techniques	–
	NIGHTs-WATCH (Mushtaq et al. 2018)		Cache leakage techniques	2%
	WHISPER (Mushtaq et al. 2020)		Cache leakage techniques	–
	(Alam et al. 2021)		Cache leakage techniques	–
	CloudRadar (Zhang et al. 2016)		Cross VM attacks	5%
	CacheShield (Briongos et al. 2018)		Cross VM attacks	–

of programs, they differ in the attack phase they target. For example, a preventive solution, called TimeWarp (Martin et al. 2012), fuzzes the timers in order to prevent attackers from making fine-grained measurements. Such fine-grained measurements are needed to distinguish between micro-architectural events like cache hits and misses. Without precise time measurements, the third phase of the attack, namely, the flush+reload, would fail. While most of these solutions are implemented in the hardware, there are also proposals that work from the software (Koruyeh et al. 2020; Wang et al. 2018; Yan et al. 2019).

Prevention at the source of leakage. These countermeasures attempt to thwart attacks at the source of leakage. The most popular approach is to redesign speculative execution in processors to make it leakage-free. A typical solution in this direction divides load instructions into safe and unsafe categories based on the threat model. For example, a load instruction that has committed its results can be considered safe, while a speculative load that is yet to be completed is considered unsafe to prevent Meltdown and Spectre-like attacks. Countermeasures designed on this philosophy allow the safe loads to alter the global state of the caches. Unsafe loads, however, are not allowed to affect the state of the cache hierarchy (Ainsworth and Jones 2020; Barber et al. 2019; González Abraham et al. 2018; Wu and Qian 2020; Yan et al. 2019). To implement this, a buffer is inserted in the processor design that temporally holds the results from speculatively executed instructions until the instruction is completed.

Prevention at the medium of leakage. Cache memories store a subset of data in the memory based on temporal and spatial locality. As the cache is several times smaller than the main memory, multiple addresses map to the same location in the cache resulting in contention. The contention is possible within a process and also across processes. An attacker models this contention to glean information in the cache, using techniques seen in section "Micro-architectural Attacks". Specialized cache memory designs have been proposed for thwarting the attacks by reducing cache contention.

In (2005), Percival suggests eliminating cache contention by modifying the cache eviction algorithms. The modified eviction algorithms would minimize the extent to which one thread can evict data from another thread. In (2005), Page proposed to partition a cache memory that was built of direct-mapped cache sets that could dynamically be partitioned into protected regions by the use of specialized cache management instructions. By tagging memory accesses with partition identifiers, each memory access is hashed to a dedicated partition. While this prevents cache contention from multiple processes, the cache memory is underutilized due to rigid partitions. For example, a process may use very few cache lines of its partition. The unused cache lines are not available to another process.

In (2007), Wang and Lee provide an improvement on the work by Page (2005) using a construct called *partition-locked cache* (PLCache), where the cache lines of interest are locked in the cache, thereby creating a private partition. These locked

cache lines cannot be evicted by other cache accesses not belonging to the private partition. In the hardware, each cache line requires additional tags comprising of a flag to indicate if the line is locked and an identifier to indicate the owner of the cache line. The underutilization of Page's partitioned cache still persists because the locked lines cannot be used by other processes, even after the owner no longer requires them.

In (2012), Domnitser et al. provide a low-cost solution to prevent attacks based on the fact that the cipher evicts one or more lines of the spy data from the cache. The solution, which requires minor modifications of the replacement policies in cache memories, restricts an application from holding more than a pre-determined number of lines in each set of a set-associative cache. With such a cache memory, the spy can never hold all cache lines in the set; therefore, the probability that the cipher evicts spy data is reduced. By controlling the number of lines that the spy can hold, a trade-off between performance and security can be achieved. Over the years, several other cache partitioning techniques have been suggested (Kiriansky et al. 2018; Sánchez and Kozyrakis 2011) which strengthens the defense while improving usability.

Another well-known modification defense for cache-based attacks makes use of randomization. Wang and Lee propose a *random-permutation cache* (RPCache) in Wang and Lee (2007), whereas the name suggests, randomizes the cache interference to make the attack more difficult. The design is based on the fact that information is leaked only when cache interference is present between two different processes. RPCache aims at randomizing such interferences so that no useful information is gleaned. The architecture requires an additional hardware called the *permutation table*, which maps the set bits in the effective address to obtain new set bits. These are then used to index the cache set array. Changing the contents of the permutation table will invalidate the respective lines in the cache. This causes additional cache misses and randomization in the cache interference.

An advancement of random cache architectures is designs that encrypt the mapping of addresses to cache sets. CEASER incorporates a block cipher (Qureshi 2018, 2019) for performing the encryption. The encryption key is periodically changed to obtain a different mapping for the cache sets. An important aspect of this design is the encryption algorithm, since it lies in the critical path and influences the time for load and store operations. While traditional ciphers have considerable latencies, ciphers designed specifically for this purpose may not provide sufficiently strong encryption (Bodduna et al. 2020).

Prevention at the transmission of leakage. While there are several techniques to thwart transient attacks by modification in the cache and the execution, existing literature also includes some preventive solution that aims to target the root cause of timing channels, such as fuzzing the timer (Martin et al. 2012) or increasing the entropy (Dhavlle et al. 2020) in the timing information. There also solutions that use program analysis (Wang et al. 2018; Zhao et al. 2020) on the program code to identify vulnerable regions and forbid speculative execution in those code sections (Fustos et al. 2019).

Detection-Based Countermeasures

Unlike prevention-based countermeasures, detection-based solutions tend to be reactive in their approach. The detection of any micro-architectural attack involves recognizing some anomalous or malicious pattern of execution. The prevalent technique to classify attacks is to discover features that can provide distinct boundaries between these attacks and benign programs using some statistical method or machine learning (ML) algorithms. Owing to this, detection-based techniques are more prudent at identifying the transmission of leakage, where the attacker performs distinct operations in the cache to glean the secrets.

A widely popular technique to capture program execution behavior is the use of Hardware Performance Counters (HPCs). These are registers provided by the hardware designer, to monitor certain micro-architectural events in the system. Originally intended for debugging purposes, over the last two decades, HPCs have been shown to profile programs to detect anomalies, malware (Demme et al. 2013), and specific micro-architectural attacks (Alam et al. 2021; Chiappetta et al. 2016; Li and Gaudiot 2019; Mushtaq et al. 2018, 2020; Zhang et al. 2016), including those based on transient execution. Such solutions do not detect the anomalies in transient execution, but the step where the attacker gleans the sensitive data.

Another approach to using HPCs for attack detection is presented by the authors in Harris et al. (2019). It uses the observation that contention in a resource leaks information only when it is cyclic, meaning domain A interferes with domain B and sequentially domain B interferes with A, thus the proposal to design a detection for such cyclic interference patterns using HPCs. While most detection techniques profile the attacker, there are approaches to profile the victim for anomalies (Briongos et al. 2018). The end goal of this design is to secure specific domains, rather than a blanket attack detection.

Conclusions

The last few years have seen several variants of transient micro-architectural attacks. The root cause in all these attacks is the unintended influence of speculatively executed operations with the hardware. Given the complexity of modern microprocessors, many new variants are likely to be discovered in the future. Next-generation microprocessors should be designed to not just prevent known attacks but should be resilient to future attacks as well. This would require security-aware design methodologies that involve the following.

- While there have been several countermeasures proposed, most have been evaluated in an ad hoc manner. This makes it difficult to quantitatively compare countermeasures and gauge their effectiveness. There is an urgent need to standardize evaluation for security in microprocessors. These standards would provide methodologies to gauge the isolation between software entities, for example, a methodology that can quantify how well the OS is isolated from

a userspace program. These methodologies could provide toolkits to analyze isolation or a suite of benchmark programs to evaluate the isolation.

- Pre- and post-silicon verification of hardware is mainly focused on functional aspects of the design. Automation tools are designed to minimize area, power, and boot performance. Security vulnerabilities, often fixed in hindsight, have proved expensive. Design automation tools should be augmented to validate for security early in the design phase. This can be a daunting task due to the vast state space of modern microprocessors. Artificial Intelligence (AI) is a promising tool that could help design automation for security. Although the use of AI in Electronic Design Automation (EDA) is in its infancy, AI is finding applications to reduce design verification time and achieve more optimized designs.
- Proposed preventive countermeasures are designed to stymie specific variants of the attacks. For example, countermeasures for Meltdown are unable to protect against the newer MDS attacks. With multiple attack variants expected in the near future, defense solutions are always catching up with the attacks.

  Detection-based countermeasures, on the other hand, can easily adapt to new attacks. However, most detection countermeasures work from software and are slow and inaccurate. CPU hardware can be augmented with attack sensors that can detect attacks at runtime with far better accuracy. These sensors should be generic enough to be configured for new attack variants.

  An alternate methodology is to use watchdogs, which monitor processor behavior to detect ongoing attacks. Programmable watchdogs have been proposed in Delshadtehrani et al. (2020) and can be extended for micro-architectural attacks.

References

Ainsworth S, Jones TM (2020) Muontrap: preventing cross-domain spectre-like attacks by capturing speculative state. In: 47th ACM/IEEE annual international symposium on computer architecture, ISCA 2020, Valencia, 30 May–3 June 2020. IEEE, pp 132–144

Alam M, Bhattacharya S, Mukhopadhyay D (2021) Victims can be saviors: a machine learning–based detection for micro-architectural side-channel attacks. J Emerg Technol Comput Syst 17(2):1–31

Barber K, Bacha A, Zhou L, Zhang Y, Teodorescu R (2019) Specshield: shielding speculative data from microarchitectural covert channels. In: 28th international conference on parallel architectures and compilation techniques, PACT 2019, Seattle, 23–26 Sept 2019. IEEE, pp. 151–164

Barresi A, Razavi K, Payer M, Gross TR (2015) CAIN: silently breaking ASLR in the cloud. In: 9th USENIX workshop on offensive technologies, WOOT'15, Washington, DC, 10–11 Aug 2015

Bernstein DJ (2005) Cache-timing Attacks on AES

Bhatkar S, DuVarney DC, Sekar R (2003) Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In: Proceedings of the 12th USENIX security symposium, Washington, DC, 4–8 Aug 2003. USENIX Association

Bhattacharyya A, Sandulescu A, Neugschwandtner M, Sorniotti A, Falsafi B, Payer M, Kurmus A (2019) Smotherspectre: exploiting speculative execution through port contention. In: Cavallaro L, Kinder J, Wang XF, Katz J (eds) Proceedings of the 2019 ACM SIGSAC

conference on computer and communications security, CCS 2019, London, 11–15 Nov 2019. ACM, pp 785–800

Bodduna R, Ganesan V, SLPSK P, Veezhinathan K, Rebeiro C (2020) Brutus: refuting the security claims of the cache timing randomization countermeasure proposed in ceaser. IEEE Comput Archit Lett 19(1):9–12

Bourgeat T, Lebedev I, Wright A, Zhang S, Arvind, Devadas S (2019) Mi6: secure enclaves in a speculative out-of-order processor. In: Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture, MICRO'52, New York. Association for Computing Machinery, pp 42–56

Briongos S, Irazoqui G, Malagón P, Eisenbarth T (2018) Cacheshield: detecting cache attacks through self-observation. In: Zhao Z, Ahn G-J, Krishnan R, Ghinita G (eds) Proceedings of the eighth ACM conference on data and application security and privacy, CODASPY 2018, Tempe, 19–21 Mar 2018. ACM, pp 224–235

Bulck JV, Minkin M, Weisse O, Genkin D, Kasikci B, Piessens F, Silberstein M, Wenisch TF, Yarom Y, Strackx R (2018) Foreshadow: extracting the keys to the intel SGX kingdom with transient out-of-order execution. In: Enck W, Felt AP (eds) 27th USENIX security symposium, USENIX security 2018, Baltimore, 15–17 Aug 2018. USENIX Association, pp 991–1008

Bulck JV, Moghimi D, Schwarz M, Lipp M, Minkin M, Genkin D, Yarom Y, Sunar B, Gruss D, Piessens F (2020) LVI: hijacking transient execution through microarchitectural load value injection. In: 2020 IEEE symposium on security and privacy, SP 2020, San Francisco, 18–21 May 2020. IEEE, pp 54–72

Canella C, Genkin D, Giner L, Gruss D, Lipp M, Minkin M, Moghimi D, Piessens F, Schwarz M, Sunar B, Bulck JV, Yarom Y (2019) Fallout: leaking data on meltdown-resistant cpus. In: Cavallaro L, Kinder J, Wang XF, Katz J (eds) Proceedings of the 2019 ACM SIGSAC conference on computer and communications security, CCS 2019, London, 11–15 Nov 2019. ACM, pp 769–784

Chen G, Chen S, Xiao Y, Zhang Y, Lin Z, Lai T-H (2020) Sgxpectre: stealing intel secrets from SGX enclaves via speculative execution. IEEE Secur Priv 18(3):28–37

Chiappetta M, Savas E, Yilmaz C (2016) Real time detection of cache-based side-channel attacks using hardware performance counters. Appl Softw Comput 49(C):1162–1174

Delshadtehrani L, Canakci S, Zhou B, Eldridge S, Joshi A, Egele M (2020) Phmon: a programmable hardware monitor and its security use cases. In: Capkun S, Roesner F (eds) 29th USENIX security symposium, USENIX security 2020, 12–14 Aug 2020. USENIX Association, pp 807–824

Demme J, Maycock M, Schmitz J, Tang A, Waksman A, Sethumadhavan S, Stolfo S (2013) On the feasibility of online malware detection with performance counters. In: Proceedings of the 40th annual international symposium on computer architecture, ISCA'13, New York. Association for Computing Machinery, pp 559–570

Dhavlle A, Mehta R, Rafatirad S, Homayoun H, Dinakarrao SMP (2020) Entropy-shield: side-channel entropy maximization for timing-based side-channel attacks. In: 21st international symposium on quality electronic design, ISQED 2020, Santa Clara, 25–26 Mar 2020. IEEE, pp 161–166

Domnitser L, Jaleel A, Loew J, Abu-Ghazaleh NB, Ponomarev D (2012) Non-monopolizable caches: low-complexity mitigation of cache side-channel attacks. TACO 8(4):35

Fustos J, Farshchi F, Yun H (2019) Spectreguard: an efficient data-centric defense mechanism against spectre attacks. In: Proceedings of the 56th annual design automation conference 2019, DAC 2019, Las Vegas, 02–06 June 2019. ACM, p 61

Gonzålez Abraham EY, Korpan B, Zhao J (2018) Spectrum: classifying , replicating and mitigating spectre attacks on a speculating risc-v microarchitecture. https://people.eecs.berkeley.edu/~kubitron/courses/cs262a-F18/projects/reports/project4_report.pdf. Accessed: 4 Apr 2021

Gras B, Razavi K, Bosman E, Bos H, Giuffrida C (2017) ASLR on the line: practical cache attacks on the MMU. In: 24th annual network and distributed system security symposium, NDSS 2017, San Diego, 26 Feb–1 Mar 2017

Harris A, Wei S, Sahu P, Kumar P, Austin TM, Tiwari M (2019) Cyclone: detecting contention-based cache information leaks through cyclic interference. In: Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture, MICRO 2019, Columbus, 12–16 Oct 2019. ACM, pp 57–72

Hund R, Willems C, Holz T (2013) Practical timing side channel attacks against kernel space ASLR. In: Proceedings of the 2013 IEEE symposium on security and privacy, SP'13. IEEE Computer Society, pp 191–205

Institute of Applied Information Processing and Communications (IAIK). Meltdown Proof-of-Concept. https://github.com/IAIK/meltdown. Accessed: 2 Mar 2021

Institute of Applied Information Processing and Communications (IAIK). ZombieLoad PoC. https://github.com/IAIK/ZombieLoad, Accessed: 2 Mar 2021

Intel. Intel C++ Compiler Classic Developer Guide and Reference. https://software.intel.com/content/dam/develop/external/documents/cpp_compiler_classic.pdf. Accessed: 3 Feb 2021

Intel Corporation (2021) 11th Generation Intel Core Processor Desktop Datasheet, Volume 1, Revision 003. https://cdrdv2.intel.com/v1/dl/getContent/634648. Accessed: 2 June 2022

Intel Corporation (2022) 12th Generation Intel Core Processor Desktop Datasheet, Volume 1, Revision 004. https://cdrdv2.intel.com/v1/dl/getContent/655258. Accessed: 2 June 2022

Khasawneh KN, Koruyeh EM, Song C, Evtyushkin D, Ponomarev D, Abu-Ghazaleh N (2019) Safespec: banishing the spectre of a meltdown with leakage-free speculation. In: Proceedings of the 56th annual design automation conference 2019, DAC'19, New York. Association for Computing Machinery

Kiriansky V, Lebedev IA, Amarasinghe SP, Devadas S, Emer JS (2018) DAWG: a defense against cache timing attacks in speculative execution processors. In: 51st annual IEEE/ACM international symposium on microarchitecture, MICRO 2018, Fukuoka, 20–24 Oct 2018. IEEE Computer Society, pp 974–987

Kocher P, Horn J, Fogh A, Genkin D, Gruss D, Haas W, Hamburg M, Lipp M, Mangard S, Prescher T, Schwarz M, Yarom Y (2019) Spectre attacks: exploiting speculative execution. In: 2019 IEEE symposium on security and privacy, SP 2019, San Francisco, 19–23 May 2019. IEEE, pp 1–19

Koruyeh EM, Khasawneh KN, Song C, Abu-Ghazaleh NB (2018) Spectre returns! speculation attacks using the return stack buffer. In: Rossow C, Younan Y (eds) 12th USENIX workshop on offensive technologies, WOOT 2018, Baltimore, 13–14 Aug 2018. USENIX Association

Koruyeh EM, Shirazi SHA, Khasawneh KN, Song C, Abu-Ghazaleh NB (2020) Speccfi: mitigating spectre attacks using CFI informed speculation. In: 2020 IEEE symposium on security and privacy, SP 2020, San Francisco, 18–21 May 2020. IEEE, pp 39–53

Li C, Gaudiot J-L (2019) Detecting malicious attacks exploiting hardware vulnerabilities using performance counters. In: Getov V, Gaudiot J-L, Yamai N, Cimato S, Chang JM, Teranishi Y, Yang J-J, Leong HV, Shahriar H, Takemoto M, Towey D, Takakura H, Elçi A, Takeuchi S, Puri S (eds) 43rd IEEE annual computer software and applications conference, COMPSAC 2019, Milwaukee, 15–19 July 2019, vol 1. IEEE, pp 588–597

Lipp M, Schwarz M, Gruss D, Prescher T, Haas W, Fogh A, Horn J, Mangard S, Kocher P, Genkin D, Yarom Y, Hamburg M (2018) Meltdown: reading kernel memory from user space. In: Enck W, Felt AP (eds) 27th USENIX security symposium, USENIX security 2018, Baltimore, 15–17 Aug 2018. USENIX Association, pp 973–990

Liu F, Lee RB (2014) Random fill cache architecture. In: 47th annual IEEE/ACM international symposium on microarchitecture, MICRO 2014, Cambridge, 13–17 Dec 2014. IEEE Computer Society, pp 203–215

Liu F, Wu H, Mai K, Lee RB (2016) Newcache: secure cache architecture thwarting cache side-channel attacks. IEEE Micro 36(5):8–16

Maisuradze G, Rossow C (2018) ret2spec: speculative execution using return stack buffers. In: Lie D, Mannan M, Backes M, Wang XF (eds) Proceedings of the 2018 ACM SIGSAC conference on computer and communications security, CCS 2018, Toronto, 15–19 Oct 2018. ACM, pp 2109–2122

Martin R, Demme J, Sethumadhavan S (2012) Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In: 39th international symposium on computer architecture (ISCA 2012), Portland, 9–13 June 2012. IEEE Computer Society, pp 118–129

Mushtaq M, Akram A, Bhatti MK, Chaudhry M, Lapotre V, Gogniat G (2018) Nights-watch: a cache-based side-channel intrusion detector using hardware performance counters. In Szefer J, Shi W, Lee RB (eds) Proceedings of the 7th international workshop on hardware and architectural support for security and privacy, HASP@ISCA 2018, Los Angeles, 02 June 2018. ACM, pp 1:1–1:8

Mushtaq M, Bricq J, Bhatti MK, Akram A, Lapotre V, Gogniat G, Benoit P (2020) WHISPER: a tool for run-time detection of side-channel attacks. IEEE Access 8:83871–83900

Page D (2005) Partitioned cache architecture as a side-channel defence mechanism. IACR Cryptology ePrint Archive 2005:280

PaX. ASLR Documentation. https://pax.grsecurity.net/docs/aslr.txt. Accessed: 2 Mar 2021

Percival C (2005) Cache missing for fun and profit. In: Proceedings of BSDCan

Qureshi MK (2018) CEASER: mitigating conflict-based cache attacks via encrypted-address and remapping. In: 51st annual IEEE/ACM international symposium on microarchitecture, MICRO 2018, Fukuoka, 20–24 Oct 2018. IEEE Computer Society, pp 775–787

Qureshi MK (2019) New attacks and defense for encrypted-address cache. In: Manne SB, Hunter HC, Altman ER (eds) Proceedings of the 46th international symposium on computer architecture, ISCA 2019, Phoenix, 22–26 June 2019. ACM, pp 360–371

Ragab H, Milburn A, Razavi K, Bos H, Giuffrida C (2021) CrossTalk: speculative data leaks across cores are real. In: S&P. Intel Bounty Reward

Ristenpart T, Tromer E, Shacham H, Savage S (2009) Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In: Proceedings of the 2009 ACM conference on computer and communications security, CCS 2009, Chicago, 9–13 Nov 2009, pp 199–212

Sakalis C, Kaxiras S, Ros A, Jimborean A, Själander M (2019) Efficient invisible speculative execution through selective delay and value prediction. In: Proceedings of the 46th international symposium on computer architecture, ISCA'19, New York. Association for Computing Machinery, pp 723–735

Sánchez D, Kozyrakis C (2011) Vantage: scalable and efficient fine-grain cache partitioning. In: Iyer R, Yang Q, González A (eds) 38th international symposium on computer architecture (ISCA 2011), San Jose, 4–8 June 2011. ACM, pp 57–68

Schunter M (2016) Intel software guard extensions: Introduction and open research challenges. In: Proceedings of the 2016 ACM workshop on Software PROtection, SPRO'16, New York. Association for Computing Machinery, p 1

Schwarz M, Lipp M, Moghimi D, Bulck JV, Stecklina J, Prescher T, Gruss D (2019a) Zombieload: cross-privilege-boundary data sampling. In: Cavallaro L, Kinder J, Wang X, Katz J (eds) Proceedings of the 2019 ACM SIGSAC conference on computer and communications security, CCS 2019, London, 11–15 Nov 2019. ACM, pp 753–768

Schwarz M, Schwarzl M, Lipp M, Masters J, Gruss D (2019b) Netspectre: read arbitrary memory over network. In: Sako K, Schneider SA, Ryan PYA (eds) Computer security – ESORICS 2019 – 24th European symposium on research in computer security, Luxembourg, 23–27 Sept 2019, Proceedings, Part I. Lecture notes in computer science, vol 11735. Springer, pp 279–299

Schwarz M, Lipp M, Canella C, Schilling R, Kargl F, Gruss D (2020) Context: a generic approach for mitigating spectre. In: NDSS

Shusterman A, Kang L, Haskal Y, Meltser Y, Mittal P, Oren Y, Yarom Y (2019) Robust website fingerprinting through the cache occupancy channel. In: 28th USENIX security symposium, USENIX security 2019, Santa Clara, 14–16 Aug 2019, pp 639–656

van Schaik S, Milburn A, Österlund S, Frigo P, Maisuradze G, Razavi K, Bos H, Giuffrida C (2019) RIDL: rogue in-flight data load. In: 2019 IEEE symposium on security and privacy, SP 2019, San Francisco, 19–23 May 2019. IEEE, pp 88–105

Wang Z, Lee RB (2007) New cache designs for thwarting software cache-based side channel attacks. In: Tullsen DM, Calder B (eds) ISCA. ACM, pp 494–505

Wang G, Chattopadhyay S, Gotovchits I, Mitra T, Roychoudhury A (2018) oo7: low-overhead defense against spectre attacks via binary analysis. ArXiv, abs/1807.05843

Wang Y, Ferraiuolo A, Zhang D, Myers AC, Edward Suh G (2016) SecDCP: secure dynamic cache partitioning for efficient timing channel protection. In: Proceedings of the 53rd annual design automation conference, DAC 2016, Austin, 5–9 June 2016. ACM, pp 74:1–74:6

Weisse O, Van Bulck J, Minkin M, Genkin D, Kasikci B, Piessens F, Silberstein M, Strackx R, Wenisch TF, Yarom Y (2018) Foreshadow-NG: breaking the virtual memory abstraction with transient out-of-order execution. Technical report

Weisse O, Neal I, Loughlin K, Wenisch TF, Kasikci B (2019) NDA: preventing speculative execution attacks at their source. In: Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture, MICRO'52, New York. Association for Computing Machinery, pp 572–586

Werner M, Unterluggauer T, Giner L, Schwarz M, Gruss D, Mangard S (2019) Scattercache: thwarting cache attacks via cache set randomization. In: Heninger N, Traynor P (eds) 28th USENIX security symposium, USENIX security 2019, Santa Clara, 14–16 Aug 2019. USENIX Association, pp 675–692

Wu Y, Qian X (2020) Reversispec: reversible coherence protocol for defending transient attacks. CoRR, abs/2006.16535

Xu J, Kalbarczyk Z, Iyer RK (2003) Transparent runtime randomization for security. In: 22nd symposium on reliable distributed systems (SRDS 2003), Florence, 6–8 Oct 2003. IEEE Computer Society, p 260

Yan M, Choi J, Skarlatos D, Morrison A, Fletcher CW, Torrellas J (2019) Invisispec: making speculative execution invisible in the cache hierarchy (corrigendum). In: Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture, MICRO 2019, Columbus, 12–16 Oct 2019. ACM, p 1076

Zhang T, Zhang Y, Lee RB (2016) Cloudradar: a real-time side-channel attack detection system in clouds. In: Monrose F, Dacier M, Blanc G, García-Alfaro J (eds) Research in attacks, intrusions, and defenses – 19th international symposium, RAID 2016, Paris, 19–21 Sept 2016, Proceedings. Lecture notes in computer science, vol 9854. Springer, pp 118–140

Zhao ZN, Ji H, Yan M, Yu J, Fletcher CW, Morrison A, Marinov D, Torrellas J (2020) Speculation invariance (invarspec): faster safe execution through program analysis. In: 53rd annual IEEE/ACM international symposium on microarchitecture, MICRO 2020, Athens, 17–21 Oct 2020. IEEE, pp 1138–1152

Bus and Memory Architectures

6

Trevor E. Carlson

Contents

Introduction 202
SoC Overview 202
Processor Overview 203
Accelerators 208
On-Chip Connectivity 209
Interconnect Interfaces 209
Interconnect Topologies 210
Off-Chip Connectivity 211
Summary and Conclusion 211
References 211

Abstract

The evolution of System-on-Chips (SoCs) is intricately linked with the efficient management of storage and connectivity within them. This chapter gives a broad overview of various SoC components, memory, and interconnect technologies. Within the communication technologies, on-chip and off-chip communication technologies are discussed, which increased in scope and complexity with the size of the SoC as well as the homogeneity/heterogeneity of the processing units. For the memory technologies, the cache management is presented with a special focus on various parameters that accurately capture the performance.

Keywords

Bus · Memory · Interconnect · Accelerator · Processor · System-on-Chip

T. E. Carlson (✉)
Department of Computer Science, National University of Singapore, Singapore, Singapore
e-mail: tcarlson@comp.nus.edu.sg

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_68

Introduction

Systems-on-Chip (SoCs) are the heart of many digital devices today. From mobile phones to TVs and smart watches to datacenter servers, most are made up of a variety of processors and accelerators connected through a variety of interconnection networks.

Connectivity is the heart of any SoC built today. These systems tend to require a collection of specialized components to handle different tasks, from external device connectivity (like display controllers, networking, and device controllers), as well as internal computation, storage, and communication (such as between compute units like CPUs (Central Processing Units), GPUs (Graphics Processing Units), and various other accelerators and internal controllers in a system). But, in addition to simple communication, there can also be isolation mechanisms for both security (ARM TrustZone ARM 2024) and performance (Bus isolation).

CPU cores, accelerators, and other peripherals of a system do not purely standalone but require interaction with other components of the system. Interestingly, one of the principles of optimization of circuits, architectures, and systems has been to reduce waste through reuse. While I can duplicate an entire processor or even a component of that processor, is there an opportunity to optimize the system if I could time-multiplex when I use those resources? This is one example of how I reduce the amount of silicon area, power, and energy used by a system. And, by optimizing the system on multiple levels, from the transistor design up to the workloads that run on the processors, it is possible to build a system that can be affordable, long-running, and efficient.

It is these very trade-offs that I will explore in this chapter. I will discuss some fundamental aspects of processor design and latency hiding methods (where the processors themselves can handle any delay caused by trade-offs made at design time). In addition to a high-level overview of the processors themselves, I will take a deep dive into SoC interconnects. The goal is to better understand how to connect various subsystems together.

SoC Overview

Today's processor designs consist of a large number of components, from the core CPU itself to accelerators like GPUs, NPUs, and various other designs. Many systems, especially embedded or power-constrained systems, build in workload-specific accelerators in ASIC form that can be used to accelerate data processing. In fact, many modern processors have been projected to contain more than 40 individual accelerators helping with a variety of tasks from decompressing audio to compressing movies for local storage (Shao and Wang 2019).

As all components connect together on the SoC, there are a number of main components that connect to the system itself. On-chip networks are used to connect components to one another on the system. Depending on connectivity requirements

Table 1 A list of the number of common digital components found in modern System-on-Chip (SoC) systems

Component	Type	Description
CPU	Compute	Latency and branch-heavy processing
GPGPU	Compute	Throughput processing on GPUs, image processing
NPU (Neural Processing Unit)	Compute	AI-specific acceleration
*PU (accelerators)	Compute	Fixed-function accelerators, like audio and video encoders and decoders
(Hierarchical) Bus interconnects	Communication	One-to-all communication
Network-on-chip	Communication	Flexible communication
On-chip memory (scratchpads) and caches	Cache hierarchy	Provides improved average latency by taking into account spatial and temporal locality
DDR interface	Off-chip communication	Off-chip DRAM communication
Peripheral interfaces	Off-chip communication	Various interfaces like HDMI, USB, and "Ethernet"

and performance requirements, solutions from a shared bus to more complex networks-on-chips are used to assure that systems can expand to meet application demands (Table 1).

In this section, I will cover how CPUs (also known as the Central Processing Unit, processor cores, or simply cores) interact with the memory hierarchy and communicate with other components on the SoC. I will also discuss memory parallelism and the structures that processors use to support various levels of memory parallelism.

Processor Overview

Processors contain a variety of components, from compute engines, like CPUs, GPUs, and AI accelerators, to communications components, like routers to support networks-on-chip.

CPUs are the most general compute element, allowing for the most diverse workloads to be executed. In addition, CPUs are usually the targets for user-compiled code, such as workloads that are run on mobile phones or datacenter servers. Alternatively, accelerators like GPUs are increasingly supported through manufacturer-supplied APIs (Application Programming Interfaces used by software) to achieve high performance and efficiency.

CPU designs have morphed significantly over the years, with enhancements coming from many areas of computer science, supporting advanced front ends,

branch-predictors, instruction prefetchers, data prefetchers, and memory aliasing optimizations. While many of these techniques affect different parts of the system in a variety of ways, the effects on how the CPU core interacts with the memory subsystem continue to be a major source of performance improvement. In fact, enhancing the CPU ability to access additional data in parallel, also known as MLP or memory-level parallelism, is a key focus of many modern CPU enhancements.

From the basic pipelined in-order processor to large, complex out-of-order processors, CPUs today support small to large numbers of outstanding requests to the memory subsystem. This was initially due to the advent of non-blocking caches with an MSHR, or Miss Status Holding Register, that allows for multiple outstanding requests from CPUs to occur at the same time.

Some workloads may be accelerated by storing recently used data close to the processor core in data caches. In this way, frequently used data can be read easily, significantly improving application performance. While data caches work when applications exhibit locality, data prefetchers help to improve core performance by watching for patterns in data accesses and repeating those patterns at a later point. The result is that there are many ways that modern computer systems aim to retrieve data in a quick and easy fashion.

But, that said, there continue to be many cases when memory accesses are difficult to predict and are not handled by many of the techniques used in systems today. Examples include graph workloads and other data-dependent workloads that contain extensive use of data-dependent conditional branches. These workloads can demonstrate significant slowdowns, even if just a few loads miss the traditional cache structures – they have a large impact on the overall performance of the workload as one cache miss to DRAM can stall the processor for an extended period of time, resulting in a significant loss of opportunity to execute instructions based on computation, not memory accesses.

There are a number of different processor types, ranging from high-efficiency, in-order processors, that handle instructions one after another. On the other extreme are out-of-order processors, which allow instructions to execute as their dependencies resolve, which is not necessarily in program order. In between, there are a variety of other processor types, from slice processors, which prioritize specific instruction types, to out-of-order commit processors, which allow instructions to commit, or finalize, early, even though it might not be in program order.

CPU Types

There are a variety of CPU types that exist (see Table 2), as mentioned in the previous section. Each processor type comes with a variety of trade-offs that sees various performance numbers, area, power, and energy-efficiency results and also results in differences in development and validation and verification time.

For extremely low-power and low-energy designs, in-order processors are typically used to control the system in an extremely lightweight way. These processors tend to be quite small (and vary in size from the extremely small nW-scale to larger mW-scale processor designs like the ARM Cortex-A520.

Table 2 A list of popular CPU types by category. Higher performance designs tend to be less efficient, while restricted out-of-order machines tend to be the most energy-efficient overall (although they are still limited in performance compared to purely out-of-order processors). In modern systems, from ARM's big.LITTLE to Intel's Efficiency and Performance cores, the complexity and efficiency of out-of-order processors can vary significantly

Processor type	Efficiency	Performance
In-order	✓✓✓	✓
Restricted out-of-order (slice-order)	✓✓	✓✓
Out-of-order	✓	✓✓✓

On the low end, processors tend to prioritize energy efficiency or area, which relate to power supply requirements or costs, respectively. For example, energy-harvesting devices that use solar power, or even electromagnetic, thermal, or vibrational energy, have very strict power and energy requirements and prioritize these requirements over others.

As performance becomes a concern, systems tend to focus on energy-efficiency and dark-silicon-compatible (Esmaeilzadeh et al. 2011) techniques to maximize power consumption given the limits seen by the physical device limits. Due to the end of Dennard Scaling (Dennard et al. 1974), the additional transistors that I typically could use for additional compute can no longer be enabled or turned on simultaneously. This has led to the need for multicore processor designs and DVFS techniques like Intel's Turbo-Boost that can allow a single processor to use a larger percentage of the overall power budget of a processor.

But, in the quest to deploy systems with continued performance improvement, CPU designers have looked to a number of performance techniques, including MLP (Glew 1998) and additional speculation in the processors in general.

Balanced Processor Architectures

Processor design is a balancing act between performance and efficiency, both power, energy and area efficiency, of the design. Processor designs consist of a number of different components that enable the execution of general-purpose workloads. Some components, like functional units and register files, are required for the proper functionality of the core. But, other structures, like branch predictors, prefetchers, caches, and a large number of other predictive structures, allow for higher performance. These structures can range in costs, but as processor designs grow, these predictive structures tend to grow as well to increase system performance.

But, these structures also provide individual design trade-offs as they scale to larger systems. Some predictive structures will need to scale linearly as the number of outstanding speculative instructions increases, while other structures might not need to scale at the same rate or even scale at all.

Determining an efficient combination of hardware structures and techniques in the processor is a balancing act, as increasing one structure to a very large size that has a low impact on performance, or will be used very infrequently, will not benefit the entire core design. Instead, by building a *balanced microarchitecture* (Karkhanis

and Smith 2007), the architect can design a system that, for the most important workloads, will not see any one structure or hardware component become the limiting factor of the system. By building a balanced microarchitecture, the resulting CPU will allow for high performance but will also not unnecessarily waste silicon area and energy for unneeded structures. By building balanced processor microarchitecture, the design can meet high performance requirements while maintaining efficiency.

CPU Memory Parallelism

Apart from the most basic of CPUs that are both in-order and are classified as stall-on-miss designs, most classes of processors today support out-of-order completion of memory accesses. Stall-on-miss processors will stall the entire CPU pipeline when the processor experiences a cache hierarchy miss. On the other hand, stall-on-use in-order processors can continue to execute until the results of the load are returned to the core. This allows the core to continue to issue loads and complete memory operations, while outstanding loads are being returned from the memory hierarchy.

Beyond in-order processors, a number of alternative processor designs exist that allow for both out-of-program-order (or just out-of-order) issue and out-of-order completion. These processors, like the slice-order processors (restricted out-of-order cores) or traditional out-of-order processors, can issue their loads once their inputs are ready (While traditional processors do require the address to be computed before issuing load instructions, for example, techniques like value prediction can predict the results of future loads and use that data to initiate future, dependent loads. While these value predicted operations are strictly speculative, they do allow for load instructions to be executed even earlier, even though the operation itself is not guaranteed to be correct. The result is higher MLP (Glew 1998) and MHP (Carlson et al. 2015), with the potential to improve system performance and efficiency further.) and are not necessarily restricted to following the original program order of the application to determine issue order.

These processors, through the use of Miss Status Holding Registers (MSHRs Kroft 1981), to be described in the following section, can accelerate the forward progress made by the processor by allowing instructions to issue early, and the data can still reach its destination.

MSHRs

MSHRs (Kroft 1981) are an important feature of caches that allow the destination of outstanding load instructions to be paired with a destination location. While the cache is waiting for the data to return, a pointer to the storage location is held in the MSHR. Once the data returns, the cache can look up the target destination (e.g., for an L1 data cache, the destination will be a register entry in the CPU) and direct the data to be stored there. Finally, the CPU uses this response as an action to eventually issue instructions that have all of their operands ready.

Apart from the L1 data cache, all caches that support a number of outstanding misses use MSHRs, including instruction caches and the caches contained in the

cache hierarchy (where modern processors can have many layers of caching, ending with the LLC, or last-level cache that will finally communicate with off-chip memory via the memory controller to request the data needed).

For a balanced CPU microarchitecture (see section "Balanced Processor Architectures"), one will size the MSHR appropriately for the system design at hand. For example, high-end in-order and low-end out-of-order processors might only have four MSHR entries in their L1 data caches. As these processors do not expose a significant amount of parallelism, a small number of MSHRs is adequate for most applications. Nevertheless, for high-end out-of-order processors, they can expose a significant amount of parallelism and tend to require 20 or more MSHRs per core.

Memory-Level and Memory Hierarchy Parallelism (MLP and MHP)

When using an advanced processor, with sufficient MSHRs and the ability to issue and receive memory instructions out of order, the processor can now request memory in parallel. MLP has been defined as the number of outstanding off-chip memory accesses, typically issued by one more processor in a system. Because of the finite speculative reach of a processor (The amount of speculative state in out-of-order processors is limited by, at least, the number of instructions that are contained in the reorder buffer (ROB) of the processor. While other structures can also limit the overall number of outstanding instructions, the limit of the number of outstanding instruction tends to be related to the size of the ROB (which includes the potential for fused instructions to extend the length further).), optimizing the number of outstanding requests can result in more real work being accomplished before the processor needs to stall for DRAM accesses to return to the processor. DRAM latencies, in the order of 10s–100s of nanoseconds, depending on the DRAM used, can easily exceed the amount of time it takes to fill an entire ROB with instructions. If the CPU is waiting for a response from DRAM and cannot process any additional instructions until the response is received, it will stall the core, preventing additional instructions from being processed.

To maximize the amount of useful work, CPU developers can aim to increase the number of simultaneous parallel memory accesses the core is capable of issuing. In this way, the processor will start additional parallel work before stalling, allowing it to reduce the total amount of time it needs to wait for DRAM responses.

For example, if a core is unable to support MLP at all (MLP is 1, or one outstanding load at a time), the core will stall waiting for responses from the memory hierarchy. Instead, if it can overlap accesses and they can be processed in parallel, the latency for all accesses will be overlapped and is approximately equal to a single DRAM access. Therefore, maximizing MLP prevents memory access serialization.

In addition to MLP, another concept called MHP, or Memory Hierarchy Parallelism, was recently introduced to explain more than off-chip parallelism, but on-chip parallelism as well (Carlson et al. 2015). This is important, as cache-level parallelism can help improve performance for high-efficiency chips that can be delayed by internal cache accesses. As a large out-of-order processor can hide most cache hits with its ability to handle a large number of outstanding instructions, smaller cores can be delayed even by cache hits to different layers of the cache

hierarchy. Many in-order and restricted out-of-order processors are unable to hide the latencies imposed by the cache hierarchy, leading to additional processor stalls that are not seen on the larger, more complex, and less efficient designs. Improving MHP for these lightweight processors can help increase throughput due to the overlapping nature of the accesses to on-chip caches (Carlson et al. 2015).

Parallelism to DRAM

Being able to access off-chip memory in a fast, high-bandwidth method is key for processors to be able to sustain high performance. The DRAM subsystem is optimized for high-bandwidth workloads, where the workloads access data in a serial manner. The DRAM structures are designed to allow for the hiding of the DRAM page activation latency and demonstrate continuous streaming of data from contiguous memory addresses. As data is streamed, a sufficiently fast CPU will be able to transfer data at the maximum rate to DRAM, which on modern machines can exceed 250 GiB/s.

MLP is enabled by the ability of the DRAM subsystem to allow accesses to different parts of the DRAM chips at a time. To handle this, DRAM is composed of banks that can handle independent requests for data. If that data is distributed across the DRAM subsystem, then there is a high chance that data accesses will need data from an unused bank. As this bank can access data independently from other banks, the accesses that do not arrive on the same bank can exhibit MLP.

Accelerators

Accelerators make up a huge portion of chips today (Shao and Wang 2019). Originally, processor designs featured a single CPU core, with basic I/O connectivity and memory (DRAM) connectivity. But, a number of factors have led to changes in how processors are designed, including the addition of multiple CPU cores and a multitude of accelerators.

The prospect of key limitations on processor performance in recent years has prevented the continued performance improvements seen in CPUs. The issue of Dark Silicon (Esmaeilzadeh et al. 2011), where the ending of Dennard Scaling (Dennard et al. 1974) prevented the use of large numbers of transistors at the same time, would lead to a requirement where only a part of the transistors on a processor could be used at once. While Moore's Law (Moore 2006) allowed for a large number of additional transistors in the same space, the power requirements to use all of these transistors at the same time were no longer possible. At this point, higher performance needed to come along with higher efficiency – leading to the broad use of specialization (in the form of accelerators), as well as more efficient cores (the use of multicore and manycore processors which swap larger inefficient cores for many smaller more efficient ones) and the use of new techniques like TurboBoost (Intel Corporation 2024b) to continue to improve processor performance (TurboBoost, and similar solutions, increases the power output of a single processor of a multicore

chip when other processors are not in use, effectively allocating power to the core in use.).

Acceleration (and specialization) has occurred along many fronts, with fixed-function accelerators (audio and video encoders are a good example of this) becoming commonplace in many designs. Off-chip accelerators (like GPUs) were the first to emerge, but on-chip acceleration quickly became dominant due to the cost savings and increased bandwidth and lower latencies of on-chip solutions. Today, mobile phone processor designs consist of a variety of accelerators and more general-purpose accelerators, from the original CPUs, GPUs, and a large number of accelerators (Shao and Wang 2019). Even CPUs have continued to specialize, increasing both performance and efficiency with SIMD, security (AES), matrix/DSP acceleration (AMX), and AI acceleration.

Together, these complex systems have a large number of discrete accelerators that need to be connected to one another in an efficient way. In the next sections, I will discuss a number of interconnect methodologies that have been developed to allow for connecting diverse components in standardized ways.

On-Chip Connectivity

On-chip interconnects provide a standard way to connect different components together in a way that provides isolation, flexibility, reusability, and ease of verification. While building custom interconnect protocols is possible and can help to reduce overheads for specific cases, generic protocols that provide functionality for specific use cases can provide most of the performance and flexibility benefits needed, without requiring the verification effort of a custom protocol.

In the following sections, I will describe a number of interconnect protocols that allow for communication between components in a standardized way.

In addition, to describe a number of common protocols used, I will also cover a number of topologies or ways to connect these components together. There are a variety of options, from a simple bus structure to a more complex network-on-chip (NoC) design. The physical topology of the system is typically abstracted from the bus connectivity. This allows for new bus topologies to be designed across generations while allowing for consistency for hardware designs.

Interconnect Interfaces

Instead of building a custom interconnect for each project or processor, I could choose to standardize on one interface (or a small number of specialized interfaces) to allow one to quickly piece together a complex SoC. In this section, I will review popular interconnect interfaces and present some of their benefits and limitations.

The Wishbone (Alderman et al. 2010) public domain interconnect interface and protocol was designed to be a general-purpose, lightweight, and open way to connect multiple processors and accelerators in a standardized way. This bus

interface supports a number of data bit widths (from 8 bits to 64 bits), as well as multiple initiators (requesting processor) and target (destination) peripherals that can exist on a single system bus. This bus design not only has support for bidirectional data transfers across a data bus but also includes optional support for many signals, like write-enable, or even the data request address. While the typical usage for an interface like this is to connect a processor to a memory, other uses like a lightweight, address-less FIFO communication protocol are also supported. The Wishbone standard also defines a protocol used for communication to initiate single data transfers, burst data transfers, read–modify–write updates, and a number of other options. The protocol allows for the ability to request a retry by the target or even to optionally support returning error results.

While there are a number of other busses that exist today, like the Avalon (Intel Corporation 2022) bus, one of the most popular interconnect types is the ARM AMBA family of bus interfaces and protocols (ARM Corporation 2024). These interfaces are royalty-free to use and come in a large variety of interface types (ARM Corporation 2024). While the basic functions of bus types like the AXI, or Advanced eXtensible Interface, bus provide similar functionality to the Wishbone bus, the signals and protocols differ. But, while the basic version of the AMBA AXI version 5 (Issue K) initiator (Manager in AMBA terminology) requires the use of 17 signals (apart from the clock and reset signals), the optional functions of AXI5 together contain more than 100 signals. Wishbone, in comparison, has just ten signals for the initiator, with more than half being optional. There are a number of advanced options that are supported by the AXI protocol, such as supporting multiple outstanding transactions and out-of-order completion of transactions (ARM Corporation 2024). These are significant enhancements, as exploiting MLP to improve performance and efficiency requires an interconnect that can support these features. A number of other interesting features are supported by the AMBA AXI protocol, such as data channels up to 1024 bits wide, atomic memory transactions, and even security support and coherence support through the CHI, or Coherent Hub Interface.

Interconnect Topologies

Interconnect topologies vary significantly due to design goals, like performance, as well as interface efficiency and complexity. One of the most basic topologies is the shared bus, where a request is broadcast from an initiator to the target to initiate an operation. While a single initiator and single target system provides a clear connection pattern, more complex connection types are allowed. For example, when there are both multiple initiators and multiple targets on the same bus, it is possible for initiators to request data and send results independently from one another.

In addition to a simple shared bus, advanced topologies can also be supported by these protocols, for example, connections that look like a ring (where initiators can connect to only two nodes, left and right), or as a 2D mesh, where connections are formed in each of the cardinal directions, north, south east, and west. These interconnect meshes can be thought of as NoCs which can use routers to achieve a

high-performance communications. Finally, the connections between busses, such as when connecting lower speed busses to high-speed ones, can be bridged to mitigate the effects of the low-speed bus on the higher speed bus.

Off-Chip Connectivity

Off-chip connectivity is an extremely important component for modern system-on-chip designs, as DRAM memory bandwidth, as one example (Cho et al. 2021), can be a limiting factor for key workloads. The key, therefore, is to maintain high-performance off-chip connectivity to allow SoCs to continue to improve in performance and efficiency.

The Peripheral Component Interconnect Express, or PCIe interface, is a common serial bus used for expansion and connectivity of high-speed devices, including GPUs and networking cards. Recent versions of PCIe include versions 6.0 (PCI-SIG) which features 256GB/s bidirectional bandwidth for 16-lane (x16) configurations and 7.0 (PCI-SIG) which increases the bandwidth to 512GB/s for the same 16-lane configuration.

DRAM connectivity typically uses the DDR (JEDEC 2008), or double data rate, bus to attach an SoC to external DRAM DIMMs (JEDEC 2023) (dual inline memory modules, the physical boards that typically the DRAM chips). Modern DDR5 (JEDEC 2024) DIMMs support up to 64GB/s per DIMM. Modern Processors (Intel Corporation 2024a) can support eight memory channels, multiplying the available bandwidth to the CPU.

Summary and Conclusion

In this chapter, I reviewed a number of important concepts with respect to the makeup of modern SoC systems. I looked at processors and how they balance traditional ILP with MLP to achieve energy-efficient performance by accelerating accesses to memory. In addition, I covered an overview of on-chip interconnects, including interfaces, protocols, and topologies. Taken together, this section provides an overview of how today's microprocessor systems have come together to enable high-performance, energy efficient designs.

References

Alderman R, Amitay Y, Cohan D, Delvaux M, Dolenc M, Hetzer V, Homann M, Hurt B, Kirk L, Lampret D, Peterson WD, Rice B, Rynearson J, Shamli A, Usselmann R, Unnebäck M, Serrano J, Wlostowski T (2010) Wishbone system-on-chip (SoC) interconnection architecture for portable IP cores, version B4

ARM (2024) Trustzone for Cortex-A. https://www.arm.com/technologies/trustzone-for-cortex-a

ARM Corporation (2024) Advanced microcontroller bus architecture (AMBA). https://developer.arm.com/Architectures/AMBA

Carlson TE, Heirman W, Allam O, Kaxiras S, Eeckhout L (2015) The load slice core microarchitecture. In: International symposium on computer architecture (ISCA), pp 272–284
Cho BY, Jung J, Erez M (2021) Accelerating bandwidth-bound deep learning inference with main-memory accelerators. In: Proceedings of the international conference for high performance computing, networking, storage and analysis, SC'21. Association for Computing Machinery, New York
Dennard RH, Gaensslen FH, Yu H, Rideout VL, Bassous E, LeBlanc AR (1974) Design of ion-implanted MOSFET's with very small physical dimensions. IEEE J Solid-State Circuits
Esmaeilzadeh H, Blem E, St. Amant R, Sankaralingam K, Burger D (2011) Dark silicon and the end of multicore scaling. SIGARCH Comput Archit News 39(3):365–376
Glew A (1998) MLP yes! ILP no. ASPLOS Wild and Crazy Idea Session
Intel Corporation (2022) Avalon®interface specifications. https://www.intel.com/content/www/us/en/docs/programmable/683091/22-3/introduction-to-the-interface-specifications.html
Intel Corporation (2024a) Intel®Xeon®6780E processor. https://ark.intel.com/content/www/us/en/ark/products/240362/intel-xeon-6780e-processor-108m-cache-2-20-ghz.html
Intel Corporation (2024b) What is Intel®Turbo Boost Technology? https://www.intel.com/content/www/us/en/gaming/resources/turbo-boost.html
JEDEC (2008) JEDEC standard: Double data rate (DDR) SDRAM, revision JESD79F. https://www.jedec.org/system/files/docs/JESD79F_0.pdf
JEDEC (2023) JEDEC standard: DDR5 unbuffered dual inline memory module (UDIMM) common standard, revision JESD308A, version 1.1. https://www.jedec.org/system/files/docs/JESD308A.pdf
JEDEC (2024) DDR5 SDRAM, revision JESD79-5C.01. https://www.jedec.org
Karkhanis TS, Smith JE (2007) Automated design of application specific superscalar processors: an analytical approach. SIGARCH Comput Archit News 35(2):402–411
Kroft D (1981) Lockup-free instruction fetch/prefetch cache organization. In: International Symposium on Computer Architecture (ISCA), pp 81–87
Moore GE (2006) Cramming more components onto integrated circuits, reprinted from electronics. IEEE Solid-State Circuits Soc Newsl
PCI-SIG. PCI-SIG®announces PCI Express®7.0 specification to reach 128 GT/s. https://www.businesswire.com/news/home/20220621005137/en
PCI-SIG. PCI-SIG®announces upcoming PCI Express®6.0 specification to reach 64 GT/s. https://www.businesswire.com/news/home/20190618005945/en/PCI-SIG-Announces-Upcoming-PCI-Express-6.0-Specification-to-Reach-64-GTs
Shao S, Wang E (2019) Die photo analysis. https://web.archive.org/web/20190518112600/http://vlsiarch.eecs.harvard.edu/research/accelerators/die-photo-analysis/

Part II
Application-Specific Processors

Architectures for Multimedia Processing: A Cross-Layer Perspective

7

Muhammad Shafique and Bharath Srinivas Prabakaran

Contents

Introduction and Overview of Video Codecs 216
High Efficiency Video Coding 218
Overview of the Standard 218
Analysis of Computational Complexity, Memory Requirements, and Processor Temperature 220
Hardware and Software Architectures for Video Coding 224
Complexity Reduction 227
Low-Power Memory Architectures 227
Workload Balancing for Multiple Video Tiles 229
Dynamic Thermal Management for HEVC 229
Future Directions 232
Conclusions 234
References 234

Abstract

Video streaming, as a domain, is one of the largest consumers of network bandwidth and accounts for more than 60% of downstream Internet traffic. As of January 2021, video streaming applications, such as Netflix, YouTube, Amazon Prime Video, HBO Max, etc., accounted for 66.2% of the global mobile data usage every month, widely overcoming other applications like social networking, web browsing, etc. This requires the research and development

M. Shafique (✉)
Engineering Division, New York University Abu Dhabi, Abu Dhabi, United Arab Emirates
e-mail: muhammad.shafique@nyu.edu

B. S. Prabakaran
Institute of Computer Engineering, Technische Universität Wien (TU Wien), Vienna, Austria
e-mail: bharath.prabakaran@tuwien.ac.at

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_7

of energy-efficient hardware and software architectures that can be deployed on handheld battery-operated devices like smartphones, to ensure that they can satisfy user requirements and application quality constraints. This chapter discusses a heterogeneous hardware-software approach that can be used to develop architectures for video coding systems, including the investigation of quality-tolerance techniques that can maximize energy efficiency.

Keywords

Multimedia · Video coding · HEVC · AVC · Accelerator · Memory · Processing · Temperature · Power · Energy · Efficiency · Hardware · Software · Architecture

Introduction and Overview of Video Codecs

Until the late 1950s, "moving pictures," or videos, could be stored only as analog signals in magnetic tapes, which were meant for playback in mechanical or cathode-ray tube-based (CRT) television (TV) systems. However, with the digital revolution, new techniques were used to create *digital videos*, which could not compete with analog videos of the time due to their infeasibly high bitrate, thereby hindering their large-scale adoption. The first generation of practical *digital videos* was made possible using a lossy compression technique called Discrete Cosine Transform (DCT). A number of companies including Toshiba and Hitachi used DCT-based algorithms to develop the first video coding standard, H.261 (Hanzo et al. 2007). Since then, DCT has been a major component of all video coding standards that followed the H.261, including the latest H.266 or Versatile Video Coding (VVC) standard (Wien and Bross 2020), established in 2020.

Even though ultra-high-definition, or 4K, is expected to be the next video standard for broadcasting services, the current generation of TV systems, from manufacturers like Samsung and LG, can already display up to 12K resolution videos. These videos require massive amounts of memory and bandwidth, in RAW formats, which make them unsuitable as a standard for video streaming or television broadcasting services. Figure 1 illustrates an overview of rising video resolutions and their corresponding memory requirement per frame in RAW format. 16K resolution videos, on average, require more than 12 million bytes per frame, which leads to the generation of over 3.3 GB of data for a 10-s video at 30 frames per second (FPS). Therefore, each new video coding standard is expected to achieve a higher level of compression to ensure that higher-resolution videos can be streamed on demand, over the Internet, or for broadcasting TV programs. The H.264 standard or Advanced Video Coding (AVC), which is currently the most-used video codec with a market share of 91% (Bitmovin 2019), was successful in reducing the bitrate by 2× when compared to its predecessor, H.263, while supporting only up to 4K resolution videos. Similarly, H.265, or High Efficiency Video Coding (HEVC), and VVC were able to further achieve ~2× compression in comparison to their

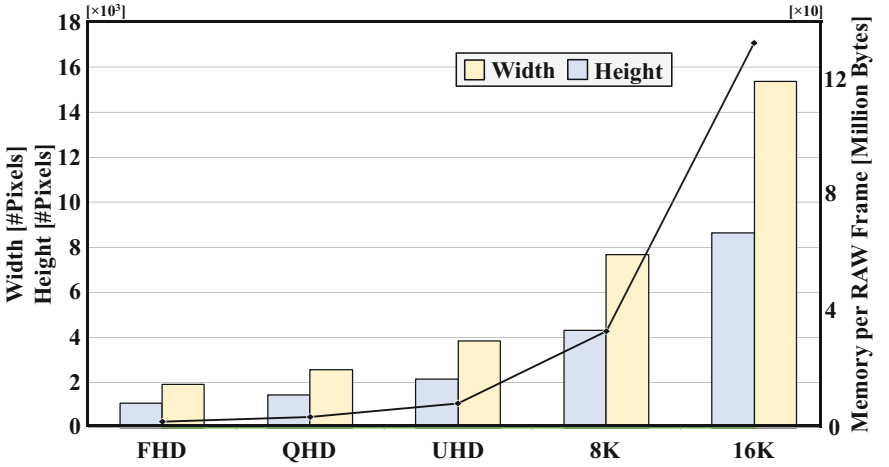


Fig. 1 Illustration of quadratically increasing memory per frame for common video resolutions

respective predecessors while supporting up to 8K and 16K video resolutions, respectively. Besides these patented video codecs, there are multiple other open-source and royalty-free alternatives like AV1, which was developed by the Alliance for Open Media (AOMedia). AOMedia is pioneered by Google and comprised of major industry partners like AMD, Broadcom, and Intel, who provide hardware support for open-source codes, while video streaming giants like Amazon and Netflix avoid spending millions of dollars in royalties every year to organizations like Technicolor and Velos Media.

Figure 2 illustrates the categorical breakdown of global downstream Internet traffic and mobile data volume. One of the largest consumers of network bandwidth, which accounts for more than 60% of downstream Internet traffic, is video streaming. As of January 2021, video streaming applications, such as Netflix, YouTube, Amazon Prime Video, HBO Max, etc., accounted for 66.2% of the global mobile data usage every month, widely overcoming other applications like social networking, web browsing, etc. This requires the research and development of energy-efficient hardware and software architectures that can be deployed on handheld battery-operated devices like smartphones and tablets, to ensure that they can satisfy user requirements and application quality constraints, which are demonstrated by primarily focusing on HEVC as a case study in this chapter. Besides HEVC, researchers have also developed a wide range of hardware and software technologies for other video coding standards such as the H.264 (Javaid et al. 2011; Khan et al. 2015; Shafique et al. 2007, 2008, 2009a,b, 2010a,b,c) and multi-view video coding (MVC) (Sampaio et al. 2013; Shafique and Zatt 2012; Shafique et al. 2012; Vizzotto et al. 2012; Zatt et al. 2011b).

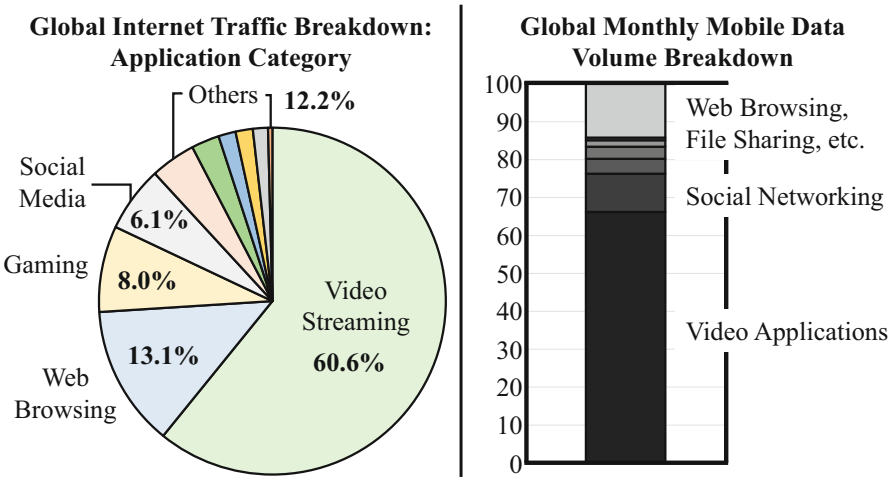


Fig. 2 An overview of the application category breakdown of global Internet traffic and mobile data volume (based on the data reported in Statista Research Department 2021 and Sandvine 2019)

High Efficiency Video Coding

Overview of the Standard

HEVC is the successor to the widely used AVC, which can achieve up to 2× better data compression or substantially improved bitrate at the same file size. The changes to HEVC include expansion of Coding Tree Units (CTUs) (the individual sub-blocks of each frame that are used for pattern and difference comparison) from 16 × 16 to 64 × 64 pixels, improved motion compensation and motion estimation, and improved variable block segmentation (Shafique and Henkel 2014). Figure 3 provides an overview of the HEVC standard and the associated operations involved in each stage. A key operation of HEVC is hybrid encoding, which involves exploiting input data redundancy by identifying inter-frame (spatial) and intra-frame (temporal) correlations using the prediction block to compress the input video stream. The I-frame, initial frame of the input stream, is encoded solely using spatial correlations and acts as a reference for future inter-frame predictions. The sum-of-absolute-difference (SAD) values for each CTU, with respect to their inter-frame and intra-frame correlations, are computed to estimate the motion vector, which can then be used for motion estimation (ME) and motion compensation. The motion vector is subsequently scaled, quantized, and transformed using the Context Adaptive Binary Arithmetic Entropy Encoder, which is transmitted to the decoder, along with the prediction information and the partitioning scheme. The picture is reconstructed by reducing the blocking artifacts using the deblocking and sample

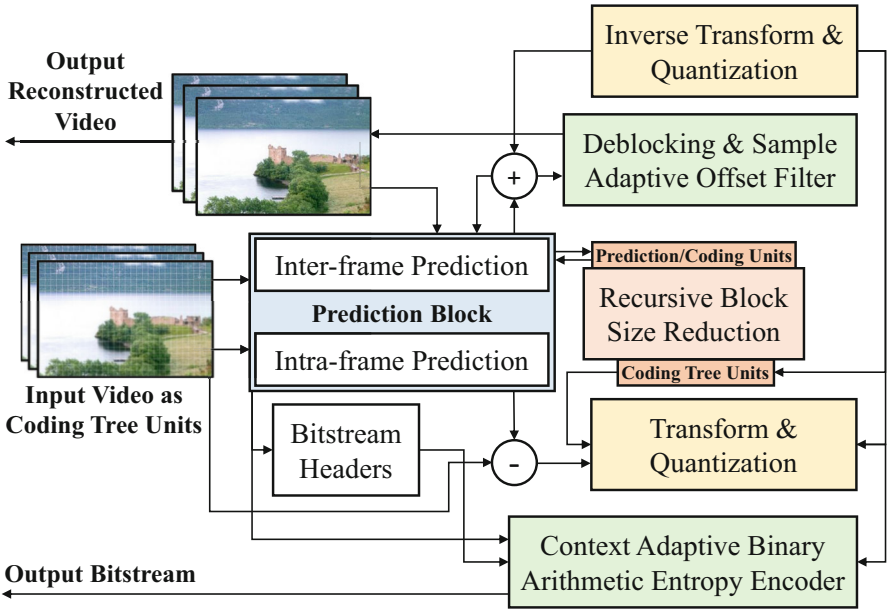


Fig. 3 Overview of the High Efficiency Video Coding (HEVC) Standard (see more details in Sullivan et al. 2012)

adaptive offset filter. The picture is subsequently stored in buffers to aid in motion-vector predictions.

Similar to the 16 × 16 blocks in AVC, HEVC deploys analogous Coding Tree Units (CTUs) composed of luma and chroma Coded Tree Blocks (CTBs). As discussed earlier, CTBs can have dimensions ranging from 16 × 16 up to 64 × 64 to ensure larger resolution video streams can be compressed appropriately. The CTBs are recursively partitioned using a quadtree structure, where each root is associated with a Coding Unit (CU), which is also partitioned into multiple Prediction (PU) and Transform Units (TU), based on whether the algorithm decides to use inter- or intra-frame prediction to encode the frame. Figure 4a illustrates an overview of the CTU partitioning into CUs, PUs, and TUs. Partitioning into PUs and TUs, or even further partitioning, may be performed at the CU level to further compress the videos. The dimensions of PU and TU can range from 64 × 64 to 4 × 4 and 32 × 32 to 4 × 4, respectively. HEVC also deploys Rate Distortion Optimization (RDO), which evaluates all combinations of CU, PU, and TU to determine the optimal partitioning that can achieve high compression efficiency. Each PU is evaluated for the 35 different intra-frame prediction modes, which are used to effectively eliminate spatial redundancy. The best prediction mode is selected by HEVC after the RDO decision at the cost of higher computational load caused by the increased search space.

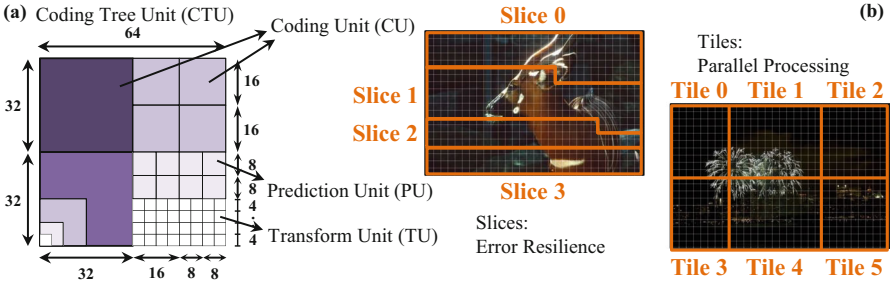


Fig. 4 (**a**) Quadtree partitioning of Coding Tree Units in HEVC; (**b**) Example of slices and tiles of a frame in HEVC for error resilience and parallel processing, respectively

However, the most complex processing stage in HEVC is the inter-frame prediction block, which is composed of motion estimation and motion compensation. This stage employs two interpolation filters that are responsible for quarter-sample precision motion compensation and fractional-pixel motion estimation. In the motion estimation stage of HEVC, the algorithm searches for a block inside the search window of the reference frame, using the SAD value to identify the motion vector. The reference frame is either stored on-chip or off-chip based on the quality-of-service requirements of the application and associated system constraints. This stage is highly memory-intensive and accounts for nearly 70% of the total energy consumed by the ME stage of HEVC (Sampaio et al. 2014).

To ensure higher error resilience, a frame can be divided into several slices (sequences of CTUs), which are raster scanned while ensuring that the predictions are not performed across slice boundaries. Tiles, on the other hand, are responsible for enabling parallel processing without any thread or workload synchronization. Figure 4b illustrates an example of slices and tiles on a video frame. Each tile is a rectangular group of CTUs, which can be independently processed on different cores of a processor without any interdependencies. However, since different CTUs of an image can vary in terms of composition and motion properties, the workload of each core may increase or decrease based on the tile assigned to the core. Therefore, the partitioning, mapping, and processing of tiles on cores and accelerators while considering the workload and its computational complexity and memory requirements are quite instrumental to the development of energy-efficient architectures and platforms.

Analysis of Computational Complexity, Memory Requirements, and Processor Temperature

The total number of predictions (β) for a CTU of dimensions M $\times$ M, provided that a CTU is partitioned into multiple CUs, is defined as:

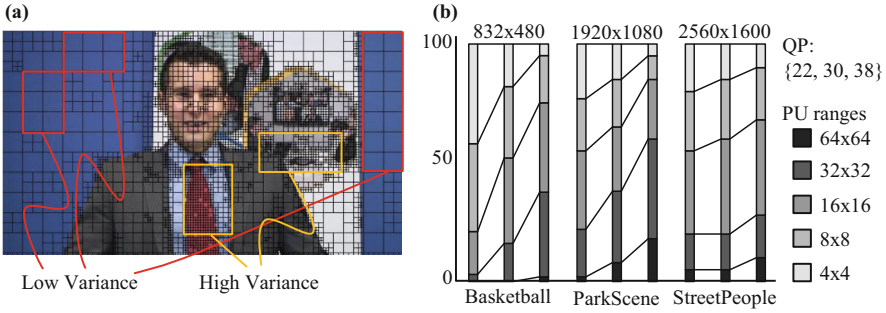


Fig. 5 (**a**) Illustration of the PUs on the initial frame of a template sequence; (**b**) percentage of image area occupied by given PU dimensions for different videos and quantization parameter values

$$\beta = \sum_{i=0}^{\log 2(M)-2} 2^{2i} \times N_i \tag{1}$$

where N_i can either denote the total number of candidates for motion search of CU size i or the total number of intra-frame prediction modes evaluated for the CTU. And as discussed earlier, all possible combinations of PUs are evaluated to identify the best CU structure for the CTU under consideration, due to the RDO. The 64×64 block structure of the CTU results in 7808 predictions for the intra-frame prediction process, which translates to roughly $2.65\times$ the number of predictions in AVC. Figure 5a illustrates the partitioning of the initial frame of a template sequence after the RDO stage in HEVC. The figure clearly identifies that larger PU dimensions are used for encoding low variance and homogeneous regions, whereas smaller blocks are used to encode high variance and heterogeneous regions. Figure 5b provides a more comprehensive breakdown of the percentage of image area occupied by different PU dimensions, quantization parameter (QP) values, and video sequences. At lower quantization levels, the texture and variations are more comprehensively captured using smaller PU blocks, with little to no large-sized PUs. Increasing the QP value, on the other hand, introduces a level of smoothing that reduces the quality of the image by increasing the number of large-sized PUs. This observation can be leveraged to research and develop an application-level complexity reduction technique that can reduce the hardware overhead of HEVC.

Vanne et al. (2012) and Bossen et al. (2012) have performed a comprehensive complexity analysis of the HEVC system to identify that three interpolation filters are responsible for 15%–38% of the total time required for encoding and decoding. These three filters are used to generate the half- and quarter-position pixels of luma and chroma eighth-pixels of the frame. Figure 6a illustrates that the interpolation filters, on average, consume 25% of the total execution time of the system, which varies depending upon the quantization parameter and the characteristics of the video sequence under consideration. Figure 6b also illustrates the variation in the

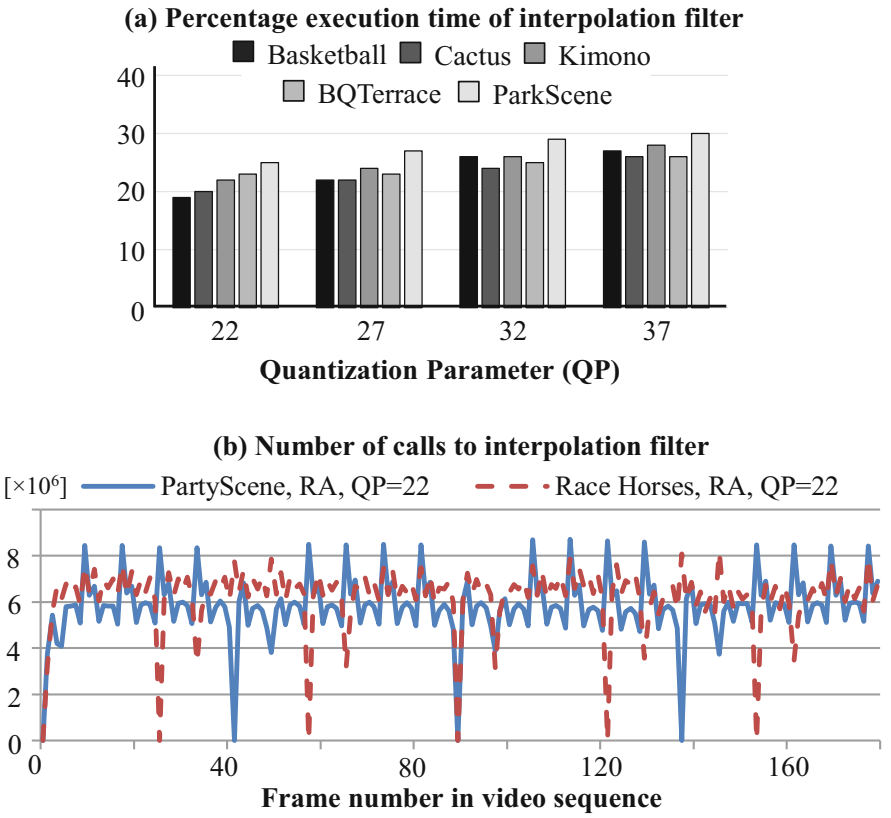


Fig. 6 (**a**) Percentage execution time of interpolation filter in HEVC encoder; (**b**) number of calls to the interpolation filter for each frame in two different video sequences. (Adapted from Shafique and Henkel 2014)

number of calls to the interpolation filters for two different video sequences, thereby motivating the need for interpolation filters that adapt with respect to the video sequence's properties.

Similar to their high computation load, the memory bandwidth of HEVC systems is, on average, 2× more when compared to the H.264 encoder, as illustrated in Fig. 7a. The box plots illustrated in Fig. 7b also denote the summary of memory access percentages for three different search window sizes (32 × 32, 64 × 64, and 128 × 128) and four video sequences. Based on this analysis, the estimated number of memory accesses of the HEVC encoder system is ~3.86× more than that of AVC. Therefore, the HEVC encoder requires more memory bandwidth and memory accesses, thereby exerting higher pressure on the memory architecture when compared to AVC, which is due to the quadtree partitioning of CTUs into multiple CUs and the video tiling processes that enable parallel processing. Figure 7b also illustrates that variation in percentage of memory accesses for

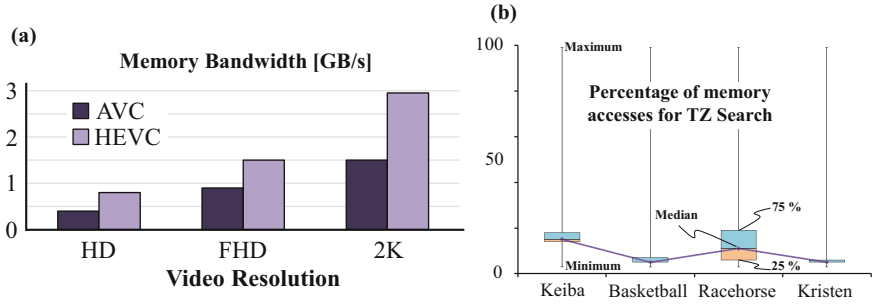


Fig. 7 (**a**) Memory bandwidth requirement for the HEVC and AVC encoder systems; (**b**) percentage statistics of block matching memory accesses for TZ search in HEVC. (Adapted from Shafique and Henkel 2014)

different workloads, i.e., video sequences. This knowledge can be leverage to develop application-specific memory hierarchies, data access and management protocols, and power-efficient designs based on the system requirements. More comprehensive analysis of HEVC systems and their requirements are presented in Khan et al. (2017) and Sampaio et al. (2014).

The temperature of the Intel processor when executing the HEVC encoder and decoder system was also analyzed using different workloads. Figure 8a illustrates the change in temperature of the processor over time when executing two different video sequences (`Keiba` and `Basketball`). The first observation is that the two workloads execute at different temperatures on the same processor after idle initial conditions, which leads us to understand that one of the workloads (`Keiba`) is more compute-intensive than the other (`Basketball`). The compute-intensive nature of `Keiba` can be attributed to the high motion content of the stream, thereby requiring a lot more computations. In the `Basketball` sequence, since the processor completes the encoding of a particular frame well before the arrival of the next frame, the processor goes into idle mode for a longer duration, thereby cooling down the chip. On average, the temperature of the processor when executing `Keiba` is 2.5°C greater than the processor temperature when `Basketball` is being executed. Curtailing high motion workloads can be used to lower the processor temperature or investigation into workload balancing techniques could be quite beneficial for balancing the processor temperature. The slack in time required for processing frames, for the low motion workloads, can be exploited to execute the workloads on low frequency cores in order to reduce the power consumption and processor temperature. Figure 8b illustrates this phenomenon by executing the `Basketball` video sequence at a much lower frequency of 1.35GHz, instead of 1.8GHz, to reduce the temperature. On the other hand, the process of video tiling, which is used to parallelize workloads, can be used to execute high motion sequences, such as `Keiba` in Fig. 8c, to meet system throughput constraints. `Keiba` is executed on two cores concurrently, which leads to a 5°C increase in the processor temperature when compared to `Basketball`.

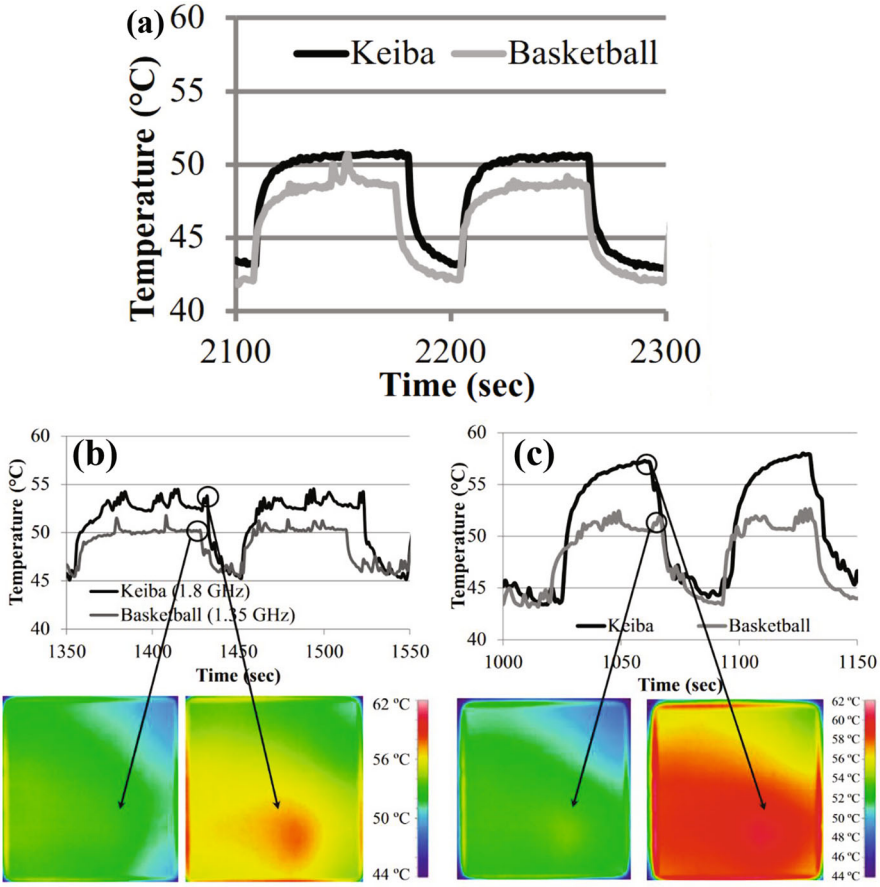


Fig. 8 (**a**) Variation in processor temperature of two different video sequences; (**b**) processor temperature analysis using frequency scaling on low motion sequence; (**c**) temperature analysis of processor using video tiling and execution on two cores. (Adapted from Shafique and Henkel 2014)

Hardware and Software Architectures for Video Coding

Figure 9 presents an overview of hardware-software architecture stack for HEVC systems. First, the input video stream is analyzed and preprocessed to characterize the data into three categories, namely, low, medium, and high, based on their motion content, texture, contrast regions, and blocks (Shafique et al. 2010a,c). Based on this categorization, workloads are budgeted for the different tiles, blocks, slices, and frames in the video stream considering the hardware capabilities of the target

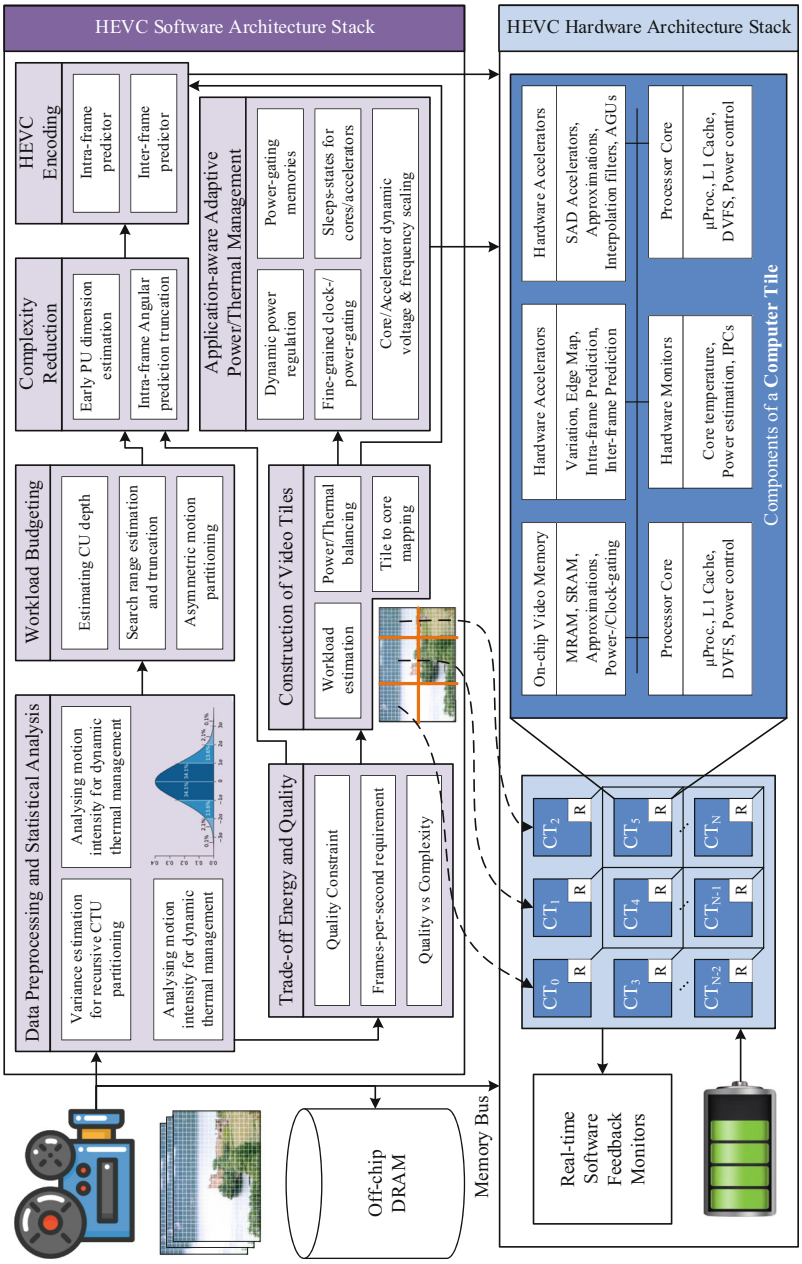


Fig. 9 An overview of the hardware-software architecture stack for HEVC systems

platform and quality constraints of the system, like the frame rate (Grellert et al. 2013). This is followed by the execution of various power management algorithms, which are used to curtail the computations of the workload and/or to identify the correct mode of operation and system configuration of the HEVC encoder that satisfies the requirements and constraints, such as the energy budget of a battery-operated device, while ensuring minimal quality loss (Khan et al. 2013a,b). Khan et al. (2014) have also proposed a video tile formation technique, which enacts intricate policies to ensure that the generated workload is balanced over different *compute tiles* (see Fig. 9 for more details).

At the hardware layer, a heterogeneous many-core architecture is envisioned where each core is denoted as a *compute tile* and is composed of (1) at least one general purpose microprocessor, which can execute the HEVC encoder software stack, (2) multiple hardware accelerators, like SAD arrays, interpolation filters, prediction blocks, etc. to ensure real-time processing and throughput (Diniz et al. 2013; Khan et al. 2013a,b; Sampaio et al. 2014; Zatt et al. 2011b), that are intertwined as co-processors on the compute tile (Diniz et al. 2013), (3) video memories with data-aware dynamic power management (DPM) capabilities (Sampaio et al. 2014; Khan et al. 2013b). Figure 10 illustrates the hardware requirements for the interpolation filters of HEVC by synthesizing them on the Xilinx XC5VLX110T-2ff1136 FPGA. The hardware techniques investigated in Sampaio et al. (2014) and Khan et al. (2013b) shift the control and power management system to the software and application layer. The software layer is supplemented with run-time hardware measures like the frame rate, throughput, and power consumption, thereby enabling workload budgeting, complexity reduction, energy-quality trade-off, etc. The inherent error resilience of the video coding application has also been exploited to design approximate accelerators for motion estimation (El-Harouni et al. 2017) and STT-RAM-based memory hierarchies (Sampaio et al. 2015; Teimoori et al. 2018) to achieve high energy efficiency.

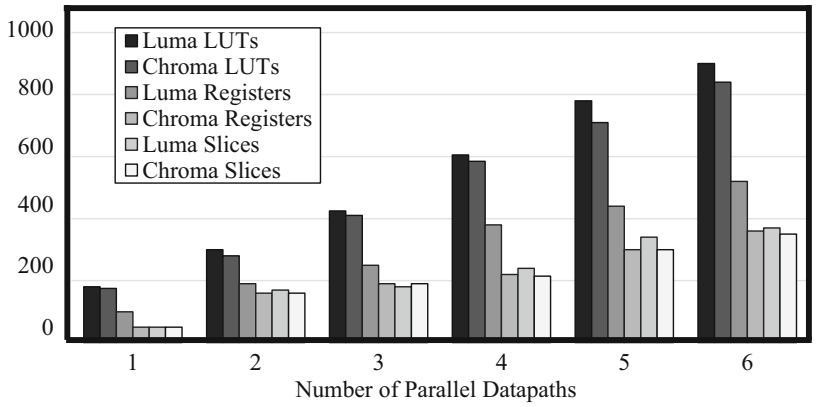


Fig. 10 Hardware results for interpolation filters on Xilinx FPGA for varying number of datapaths

Complexity Reduction

Limiting the total number of partitions for CU can drastically reduce the complexity of the HEVC encoder. Based on a preliminary analysis of input video stream that identifies the variance properties of different object, the approach successfully determines a set of CU sizes most suitable for the scenario, thereby reducing the overhead of the full RDO mode decision. Based on the preliminary analysis illustrated in section "Analysis of Computational Complexity, Memory Requirements, and Processor Temperature", which discussed the dependence of CU dimensions on the variance of the input stream, a variance-aware CU/PU mode prediction technique has been developed. Initially, the variance of the CTU, at the 4×4 sublevel, is estimated. This information is used to recursively merge similar neighboring blocks, based on their variance, to create bigger blocks and partitions. This process is repeated until the point where there are no more blocks left to be merged, at which point we are left with a "PU Map," which is used for evaluating the PU size predictions. This also eliminates the need for multiple CU/PU dimension evaluation as the PU map dimension evaluations prove to be sufficient in most cases. To further lower the risk of misprediction, additional PU map is generated, "PU Map Above," which denotes the PU map a level above the current node in the quadtree partitioning structure, as shown in Fig. 11. The approach is successful in achieving a speedup of 44%, on average, and energy reductions of 35% while incurring a negligible loss of −0.048 dB output quality (BD-PSNR (Bjontegaard 2001)). This complexity reduction technique is orthogonal to other complexity reduction mechanisms like early-stage TU/PU partitioning and intra-frame angular direction prediction.

Low-Power Memory Architectures

As discussed in section "Analysis of Computational Complexity, Memory Requirements, and Processor Temperature", HEVC is a highly memory-intensive application, which requires the devices to store the current and multiple reference frames in order to perform motion search and block matching. This problem is further aggravated by higher-resolution videos. This problem can be addressed by analyzing the memory access pattern of the HEVC encoder across multiple

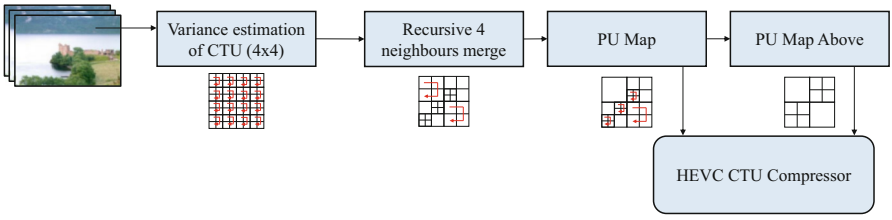


Fig. 11 Overview of the approach used to select the best CU dimensions and locations

input video streams. For instance, large reference video frames are repeatedly loaded to on-chip memory during the motion search process, which leads to large power consumption of the on-chip memories and the power consumption caused by repeated memory accesses. These problems can be either addressed by (1) reducing the number of repeated off-chip memory accesses by exploring effective data reuse technique, motion trajectory prediction, which can be exploited to design and manage application-specific on-chip memories (Sampaio et al. 2014; Shafique and Henkel 2011; Zatt et al. 2011a,b), or (2) leveraging next-generation memory technologies that overcome the volatility limitations of SRAM, such as MRAM, which drastically reduce the leakage power of the cell by ~5.4× while incurring a 2.6× higher write latency and nearly 20× higher dynamic power during the write cycle.

Instead of exploring these two approaches individually, Khan et al. (2013b) proposed to combine the benefits of both approaches by designing a hybrid memory architecture that combines SRAM and NVMs with adaptive energy management system called AMBER, as illustrated in Fig. 12a. To enable low read latency of CTUs and hide the incredibly long latency of MRAM writes, a small SRAM-based memory is included on-chip. The CTUs fetched from off-chip memory are simultaneously written to the SRAM-based memory and MRAM, which can be subsequently used for repeated future accesses during the motion estimation stage. The MRAM buffers are divided into multiple sectors on the chip, which can be individually clock-/power-gated based on the memory requirement of the video stream/application, to reduce power consumption. Due to its non-volatility, MRAMs retain the data when power-gated, making them highly suitable for such hybrid memory architectures. The proposed power manager is a self-organizing map-based learner, which adapts itself, without any supervision, to changing memory patterns and effectively reduces power consumption when compared to the state of the art, as illustrated in Fig. 12b. However, since AMBER does not support the bandwidth required for processing multiple tiles in parallel with concurrent memory access, Sampaio et al. (2014) proposed a *distributed scratchpad-based video memory* for HEVC. This technique employs a data-driven dynamic power management tool that reduces energy consumption by up to 61% compared to

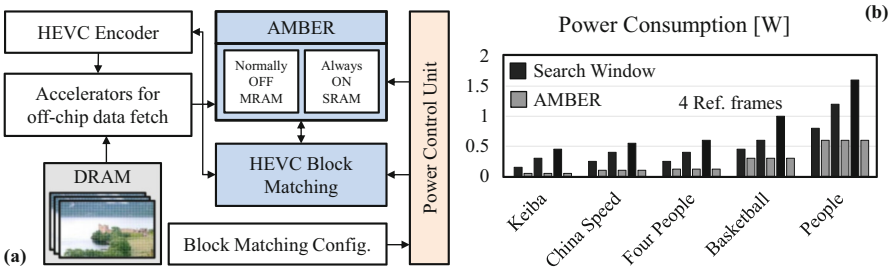


Fig. 12 (**a**) Overview of the proposed AMBER memory system; (**b**) illustration of power savings achieved by the AMBER system

state-of-the-art solutions while also reducing the on-chip energy leakage by 54% by incorporating the application-specific knowledge.

Workload Balancing for Multiple Video Tiles

As discussed earlier, video tiles can be used to effectively parallelize HEVC by encoding independent video tiles on different cores/compute tiles, concurrently, without the need for thread synchronization. Furthermore, as discussed earlier and illustrated in Fig. 13a, the workload for different tiles varies based on the composition and content of their CTUs. Therefore, the construction of video tiles is an important problem that needs to be addressed by considering the hardware resources of the target platform and the application-specific requirements. For example, in case the number of available processing cores and their maximum operating frequency information are not considered, the power budget could be exceeded or the performance of the system might not satisfy the user. Moreover, increasing the number of tiles leads to a loss in video quality, which needs to be aptly considered while addressing the tiling problem to find an appropriate solution.

Figure 13b illustrates an overview of the workload balancing technique for parallelizing the HEVC system. First, the complete workload is equally divided among all available cores. In case the number of cores is higher, all additional cores are power-gated. Next, the required quality and throughput metrics are evaluated to determine if the system satisfies the application constraints. If not, the complexity reduction technique presented in section "Complexity Reduction" is deployed to curtail the maximum workload for each core before encoding begins. Based on the analysis, video tiles in a given frame, typically, have a high spatial correlation and temporal correlation with collocated video tiles. This information is exploited to estimate the time required to process and generate the upcoming frame. Since encoded video quality can be considered as another criterion, the number of tiles in the frame can be set by the user as part of the quality requirement. The power savings obtained in this work satisfy the user-specific tolerance level (more details in Khan et al. 2014). Figure 13c and d illustrate the bitrate, frequency, and total intra-frame modes for two different tiles of the `Exit` sequence.

Dynamic Thermal Management for HEVC

To circumvent the temperature problem in HEVC systems (see discussion in section "Analysis of Computational Complexity, Memory Requirements, and Processor Temperature"), Palomino et al. (2014) have proposed an application-aware dynamic thermal management (DTM) policy. The policy requires a set of Pareto-optimal configurations which trade off between the temperature of different coding configurations and their associated video quality (in PSNR). The DTM first collects baseline temperature values (T_C) from the sensor, to determine if the value could reach a predetermined thermal threshold. In case it can, the DTM chooses an

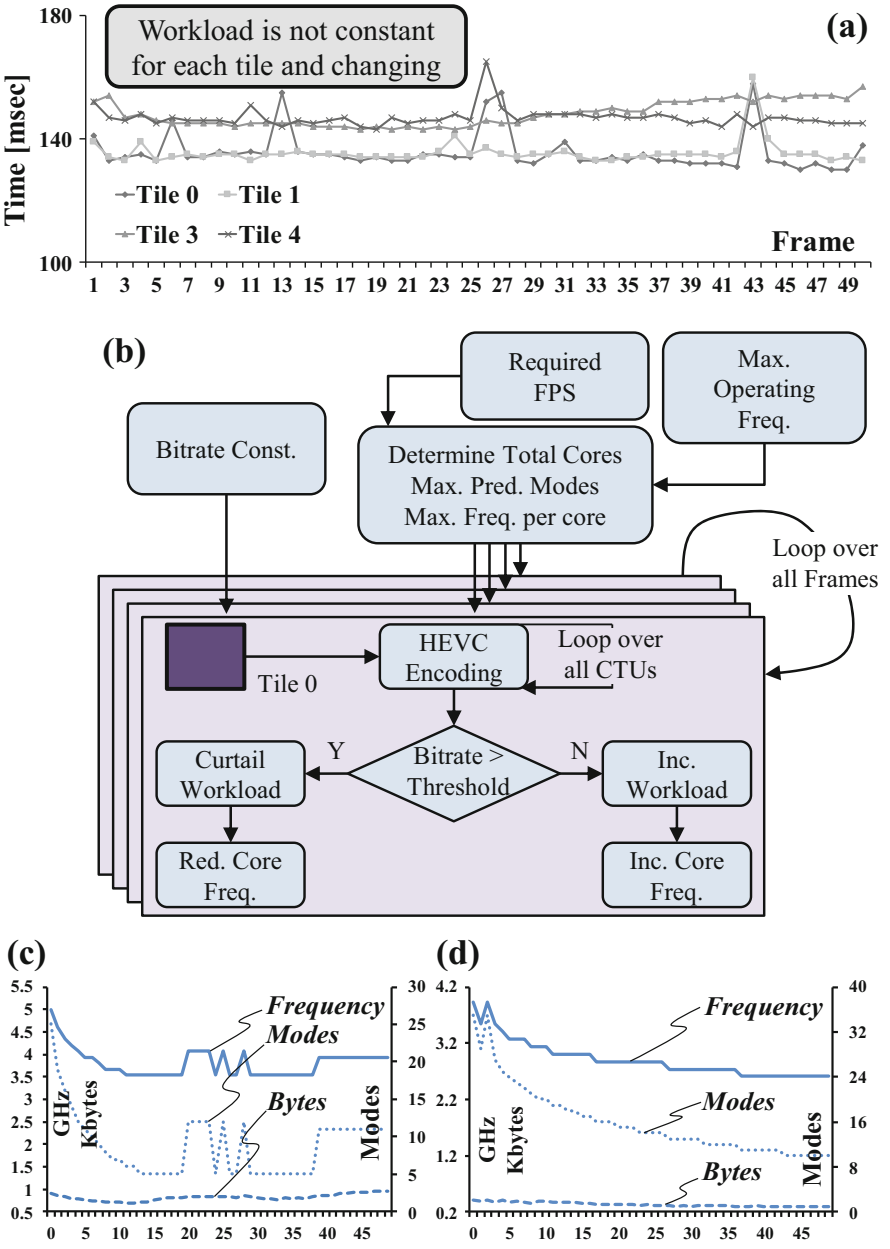


Fig. 13 (**a**) Time required for processing each tile with four tiles per frame using the `RaceHorses`; (**b**) the adaptive workload balancing technique for many-core processors; (**c**) adaptation analysis for two different tiles in `Exit`. (Adapted from the data presented in Shafique and Henkel 2014)

appropriate configuration from the set of Pareto-optimal configurations, followed by fine-tuning the configuration at frame and CU level based on motion and texture information (Shafique et al. 2010a,c). In case of thermal emergencies ($T_C > T_{th}$), the DTM triggers voltage and frequency scaling to immediately reduce the power to the core, thereby reducing its temperature. The efficacy of the approach is illustrated in Fig. 14a, wherein the maximum, average, and minimum temperatures of the DTM policy are well below the "No DTM" approach. This is even more prominent when the threshold is decreased substantially. The algorithm is initiated with the coding configuration that offers the best quality, but might soon be replaced by the DTM to ensure that thermal constraints are met, which might, in turn, reduce the output quality. At $T_{th} = 54\,°C$, the approach incurs a negligible quality loss of 0.007 dB. Under tight temperature constraints, such as $T_{th} = 46\,°C$, the quality loss might be significant and noticeable. A more comprehensive analysis of the DTM approach is presented in Fig. 14b and c, which illustrates a frame-level breakdown of the temperature profile used for encoding the `Keiba` and `Basketball` sequences. In the No DTM scenario, the chip constantly registers temperatures above 52 °C generating unfavorable thermal maps and heating profiles. However, the DTM-

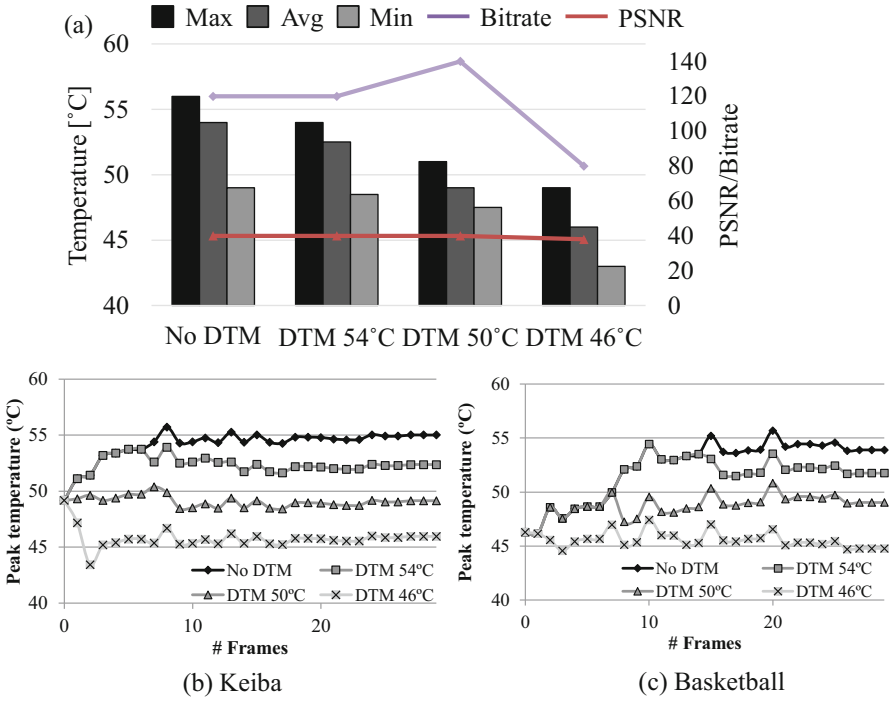


Fig. 14 (**a**) Maximum, average, and minimum processor temperature for encoding `Keiba`; (**b**) comparison of temperature profile for DTM and non-DTM approaches. (Adapted from Shafique and Henkel 2014)

based approaches significantly reduce the average temperature to ensure long-term device usage.

Open-Source Multi-threaded HEVC Software Architecture: The current reference encoders require a lot of memory and are quite slow to execute, which immediately dampens their portability to embedded systems. Due to their serial implementation in the original version of the codec or any other version of the H.265, the software infrastructure lacks parallelization, which can take advantage of the underlying multi-core processor system. Toward this, the authors have also developed an open-source multi-threaded version of the HEVC intra-frame prediction-only encoder (Khan et al. 2014). The complete framework was implemented in C++, which can be compiled individually on Windows and Linux platforms for testing, research, and development. To ensure reproducibility and ease of research, the project is open-source and available online at https://ces265.sourceforge.io. The software architecture allows slice- and tile-level threading capabilities which can be exploited to achieve up to ~13.2× faster execution compared to the HM-9.2 reference software.

Future Directions

Since its inception, "moving pictures" have been consumed by users in a variety of ways, starting from film projectors in the 1900s all the way up to in-home streaming using on-demand over-the-top (OTT) media service providers like Netflix, YouTube, Amazon Prime, etc. These OTT ventures, which were growing steadily due to their on-demand accessibility and comfort of home use, started registering exponential growth when the COVID-19 pandemic hit and people were confined to their homes without any other entertainment options (World Health Organization 2021). For instance, the number of paid OTT subscribers in India increased by 30% to more than 30 million in a span of 5 months, from March to July 2020, when lockdown was instituted in the country (Financial Express 2020). Similar trends were observed in almost all the countries where extended periods of lockdown were instituted in order to reduce the spread of the novel coronavirus (Matthew Ball 2020). These OTT ventures, which rely on H.264 and HEVC video standards to transmit the high-resolution videos, require hardware-level acceleration support and high bandwidth requirements to enable real-time streaming, as discussed in section "Introduction and Overview of Video Codecs". These requirements are increasingly important when the next generation of video coding standards is devised as they are likely to require higher computational power and bandwidth requirements, such as the VVC standard.

Besides on-demand entertainment, videos are dominating domains like virtual and augmented reality, which are encoded to achieve 360-degree videos (Bross et al. 2021; Zhou et al. 2019), gaming (Zadtootaghaj et al. 2018), video confer-

encing (Wang et al. 2021), and many other applications. Each of these applications is accompanied by research challenges highly specific to the application use case. For example, the hardware resources available on the VR headset should be capable of rendering 360-degree 4K resolution videos at 60+ FPS in real time while being compact and not consume too much power or generate a lot of heat, which can cause discomfort to the user. Similarly, the quality and frame rate requirement of online gaming platforms are satisfied using specialized Graphics Processing Units (GPUs), which can handle the memory bandwidth and highly parallelized compute requirements of sophisticated gameplay that changes in real time based on user controls. The next steps to realize such ventures would be to investigate, research, and develop hardware-software solutions to meet the requirements of such use cases, potentially using a modified version of the methodology presented in section "Hardware and Software Architectures for Video Coding".

More recently, quite a few of the scientific advancements in computing, like near-memory computing (Singh et al. 2019) and deep learning (LeCun et al. 2015), have also had major impacts on the video coding domain, which have furthered capabilities across various computing fields. Lesniak et al. (2021) have proposed a novel high-bandwidth memory (HBM) interfaced processor architecture that can enable hardware-accelerated execution for applications like video coding and encryption, which require high memory bandwidth. Besides HBMs, Resistive Random-access Memories (ReRAM) and associated cross-bar architectures have proven to be quite beneficial in performing in-memory computing for large-scale data processing applications like deep learning (Chi et al. 2016). These architectures could also be quite useful for video coding applications once the cost and performance benefits associated with their fabrication and their data reliability challenges are addressed.

Likewise, the deep learning paradigm has enabled a whole sleuth of advances in the computer vision domains, which has enabled a wide range of features in autonomous driving, robotics, etc. For instance, Deep Neural Networks have proven to be state of the art in Computer Vision (like image detection, recognition, and segmentation), Natural Language Processing (like speech recognition, machine translation, and sentiment analysis), Medical Assistance (tumor segmentation, diagnostics, and drug discovery), and many other applications. Deep learning has also invaded the video coding domain; Wang et al. (2021) have leveraged the capabilities of Deep Neural Networks and their proposed keypoint representation technique to reduce the bandwidth requirement for video conferencing applications by 10×, when compared to the H.264 video coding standard. Their technique can also be used to synthesize a pseudo-realistic face-to-face experience during video conferencing.

Therefore, it is in the best interest of researchers worldwide to investigate novel and upcoming scientific advancements to address the challenges associated with the video coding domain and its applications in fields like video archiving, OTT content streaming, video surveillance, computer vision, robotics, autonomous driving, human-computer interaction, etc. to improve the quality of service.

Conclusions

After a thorough analysis of the computational complexity, memory requirement, and temperature concerns of HEVC systems, the authors presented a comprehensive hardware-software approach, which considers the application properties and context to discuss relevant approaches that can be used to increase the system efficacy. A low-power HEVC system is presented in this chapter, which exploits content-specific knowledge in both the hardware and software layers by means of (1) an adaptive complexity reduction technique, (2) a low-power many-core processor architecture with compute tiles, (3) a hybrid video memory with dynamic power management, (4) a video tile construction and workload balancing approach, and (5) a dynamic thermal management technique (for more information on these topics, the authors refer the readers to Diniz et al. 2013, Grellert et al. 2013, Khan et al. 2013a,b, 2014, Sampaio et al. 2014, Shafique et al. 2010a,c, and Zatt et al. 2011b). Similarly, a more complicated and complex system like the H.266, or Versatile Video Coding Standard, has a very high potential for research in enabling power-efficient hardware and software architectures for multimedia playback.

Acknowledgments We would like to explicitly thank Felipe Sampaio, Bruno Zatt, Sergio Bampi, Daniel Palomino, Muhammad Usman Karim Khan, and Jörg Henkel for their contributions to parts of the works cited in this chapter. We would also like to thank other researchers in industry and academic alike, especially the ones cited in this work, who contributed to this field to enable advancements that helped us realize the potential of video coding across multiple domains.

References

Bitmovin (2019) Video developer report. https://bitmovin.com/bitmovin-2019-video-developer-report-av1-codec-ai-machine-learning-low-latency/

Bjontegaard G (2001) Calculation of average PSNR differences between RD-curves. VCEG-M33

Bossen F, Bross B, Suhring K, Flynn D (2012) HEVC complexity and implementation analysis. IEEE Trans Circuits Syst Video Technol 22(12):1685–1696

Bross B, Chen J, Ohm JR, Sullivan GJ, Wang YK (2021) Developments in international video coding standardization after AVC, with an overview of versatile video coding (VVC). In: Proceedings of the IEEE

Chi P, Li S, Xu C, Zhang T, Zhao J, Liu Y, Wang Y, Xie Y (2016) Prime: a novel processing-in-memory architecture for neural network computation in reram-based main memory. ACM SIGARCH Comput Archit News 44(3):27–39

Diniz CM, Shafique M, Bampi S, Henkel J (2013) High-throughput interpolation hardware architecture with coarse-grained reconfigurable datapaths for HEVC. In: 2013 IEEE international conference on image processing. IEEE, pp 2091–2095

El-Harouni W, Rehman S, Prabakaran BS, Kumar A, Hafiz R, Shafique M (2017) Embracing approximate computing for energy-efficient motion estimation in high efficiency video coding. In: Design, automation & test in Europe conference & exhibition (DATE), 2017. IEEE, pp 1384–1389

Financial Express (2020) Rise of paid subscribers. https://www.financialexpress.com/brandwagon/2020-rise-of-paid-subscribers/2172942/

Grellert M, Shafique M, Khan MUK, Agostini L, Mattos JC, Henkel J (2013) An adaptive workload management scheme for HEVC encoding. In: 2013 IEEE international conference on image processing. IEEE, pp 1850–1854

Hanzo L, Cherriman P, Streit J (2007) Video compression and communications: from basics to H. 261, H. 263, H. 264, MPEG4 for DVB and HSDPA-style adaptive turbo-transceivers. Wiley, Hoboken

Javaid H, Shafique M, Parameswaran S, Henkel J (2011) Low-power adaptive pipelined mpsocs for multimedia: an H. 264 video encoder case study. In: 2011 48th ACM/EDAC/IEEE design automation conference (DAC). IEEE, pp 1032–1037

Khan MUK, Shafique M, Grellert M, Henkel J (2013a) Hardware-software collaborative complexity reduction scheme for the emerging HEVC intra encoder. In: 2013 design, automation & test in Europe conference & exhibition (DATE). IEEE, pp 125–128

Khan MUK, Shafique M, Henkel J (2013b) An adaptive complexity reduction scheme with fast prediction unit decision for HEVC intra encoding. In: 2013 IEEE international conference on image processing. IEEE, pp 1578–1582

Khan MUK, Shafique M, Henkel J (2014) Software architecture of high efficiency video coding for many-core systems with power-efficient workload balancing. In: 2014 design, automation & test in Europe conference & exhibition (DATE). IEEE, pp 1–6

Khan MUK, Shafique M, Bauer L, Henkel J (2015) Multicast fullhd H. 264 intra video encoder architecture. IEEE Trans Comput-Aided Des Integr Circuits Syst 34(12):2049–2053

Khan MUK, Shafique M, Henkel J (2017) Energy efficient embedded video processing systems: a hardware-software collaborative approach. Springer, Berlin

LeCun Y, Bengio Y, Hinton G (2015) Deep learning. Nature 521(7553):436–444

Lesniak F, Kreß F, Becker J (2021) Transparent near-memory computing with a reconfigurable processor. In: International symposium on applied reconfigurable computing. Springer, pp 221–231

Matthew Ball (2020) The impact of COVID-19 on pay-tv and ott video. https://www.matthewball.vc/all/covidvideo

Palomino D, Shafique M, Amrouch H, Susin A, Henkel J (2014) HEVCDTM: application-driven dynamic thermal management for high efficiency video coding. In: 2014 design, automation & test in Europe conference & exhibition (DATE). IEEE, pp 1–4

Sampaio F, Zatt B, Shafique M, Agostini L, Henkel J, Bampi S (2013) Content-adaptive reference frame compression based on intra-frame prediction for multiview video coding. In: 2013 IEEE international conference on image processing. IEEE, pp 1831–1835

Sampaio F, Shafique M, Zatt B, Bampi S, Henkel J (2014) DSVM: energy-efficient distributed scratchpad video memory architecture for the next-generation high efficiency video coding. In: 2014 design, automation & test in Europe conference & exhibition (DATE). IEEE, pp 1–6

Sampaio F, Shafique M, Zatt B, Bampi S, Henkel J (2015) Approximation-aware multi-level cells STT-RAM cache architecture. In: 2015 international conference on compilers, architecture and synthesis for embedded systems (CASES). IEEE, pp 79–88

Sandvine (2019) Global internet phenomena report. https://www.sandvine.com/press-releases/sandvine-releases-2019-global-internet-phenomena-report

Shafique M, Henkel J (2011) Hardware/software architectures for low-power embedded multimedia systems. Springer Science & Business Media, Berlin

Shafique M, Henkel J (2014) Low power design of the next-generation high efficiency video coding. In: 2014 19th Asia and South Pacific design automation conference (ASP-DAC). IEEE, pp 274–281

Shafique M, Zatt B (2012) A complexity reduction scheme with adaptive search direction and mode elimination for multiview video coding. In: 2012 picture coding symposium. IEEE, pp 105–108

Shafique M, Bauer L, Henkel J (2007) An optimized application architecture of the H. 264 video encoder for application specific platforms. In: 2007 IEEE/ACM/IFIP workshop on embedded systems for real-time multimedia. IEEE, pp 119–124

Shafique M, Bauer L, Henkel J (2008) 3-tier dynamically adaptive power-aware motion estimator for H. 264/avc video encoding. In: Proceeding of the 13th international symposium on low power electronics and design (ISLPED'08). IEEE, pp 147–152

Shafique M, Bauer L, Henkel J (2009a) A parallel approach for high performance hardware design of intra prediction in H. 264/avc video CODEC. In: 2009 design, automation & test in Europe conference & exhibition. IEEE, pp 1434–1439

Shafique M, Molkenthin B, Henkel J (2009b) Non-linear rate control for H. 264/avc video encoder with multiple picture types using image-statistics and motion-based macroblock prioritization. In: 2009 16th IEEE international conference on image processing (ICIP). IEEE, pp 3429–3432

Shafique M, Bauer L, Henkel J (2010a) enbudget: a run-time adaptive predictive energy-budgeting scheme for energy-aware motion estimation in H. 264/mpeg-4 avc video encoder. In: 2010 design, automation & test in Europe conference & exhibition (DATE 2010). IEEE, pp 1725–1730

Shafique M, Bauer L, Henkel J (2010b) Optimizing the H. 264/avc video encoder application structure for reconfigurable and application-specific platforms. J Sig Process Syst 60(2):183–210

Shafique M, Molkenthin B, Henkel J (2010c) An HVS-based adaptive computational complexity reduction scheme for H. 264/avc video encoder using prognostic early mode exclusion. In: 2010 design, automation & test in Europe conference & exhibition (DATE 2010). IEEE, pp 1713–1718

Shafique M, Zatt B, Walter FL, Bampi S, Henkel J (2012) Adaptive power management of on-chip video memory for multiview video coding. In: DAC design automation conference 2012. IEEE, pp 866–875

Singh G, Chelini L, Corda S, Awan AJ, Stuijk S, Jordans R, Corporaal H, Boonstra AJ (2019) Near-memory computing: past, present, and future. Microprocess Microsyst 71:102868

Statista Research Department (2021) Global mobile data traffic share. https://www.statista.com/statistics/383715/global-mobile-data-traffic-share/

Sullivan GJ, Ohm JR, Han WJ, Wiegand T (2012) Overview of the high efficiency video coding (HEVC) standard. IEEE Trans Circuits Syst Video Technol 22(12):1649–1668

Teimoori MT, Hanif MA, Ejlali A, Shafique M (2018) Adam: adaptive approximation management for the non-volatile memory hierarchies. In: 2018 design, automation & test in Europe conference & exhibition (DATE). IEEE, pp 785–790

Vanne J, Viitanen M, Hamalainen TD, Hallapuro A (2012) Comparative rate-distortion-complexity analysis of HEVC and avc video codecs. IEEE Trans Circuits Syst Video Technol 22(12):1885–1898

Vizzotto BB, Zatt B, Shafique M, Bampi S, Henkel J (2012) A model predictive controller for frame-level rate control in multiview video coding. In: 2012 IEEE international conference on multimedia and expo. IEEE, pp 485–490

Wang TC, Mallya A, Liu MY (2021) One-shot free-view neural talking-head synthesis for video conferencing. In: Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, pp 10039–10049

Wien M, Bross B (2020) Versatile video coding–algorithms and specification. In: 2020 IEEE international conference on visual communications and image processing (VCIP). IEEE, pp 1–3

World Health Organization (2021) COVID-19 weekly epidemiological update, edition 46, 29 june 2021. https://www.financialexpress.com/brandwagon/2020-rise-of-paid-subscribers/2172942/

Zadtootaghaj S, Schmidt S, Barman N, Möller S, Martini MG (2018) A classification of video games based on game characteristics linked to video coding complexity. In: 2018 16th annual workshop on network and systems support for games (NetGames). IEEE, pp 1–6

Zatt B, Shafique M, Bampi S, Henkel J (2011a) A low-power memory architecture with application-aware power management for motion & disparity estimation in multiview video coding. In: 2011 IEEE/ACM international conference on computer-aided design (ICCAD). IEEE, pp 40–47

Zatt B, Shafique M, Sampaio F, Agostini L, Bampi S, Henkel J (2011b) Run-time adaptive energy-aware motion and disparity estimation in multiview video coding. In: 2011 48th ACM/EDAC/IEEE design automation conference (DAC). IEEE, pp 1026–1031

Zhou Y, Tian L, Zhu C, Jin X, Sun Y (2019) Video coding optimization for virtual reality 360-degree source. IEEE J Sel Top Sig Process 14(1):118–129

Post-Quantum Cryptographic Accelerators

8

Ayesha Khalid and Dur-e-Shahwar Kundi

Contents

Introduction . . . 238
Post-Quantum Cryptography (PQC) . . . 240
NIST Post-Quantum Cryptography Standardisation Project . . . 240
Classes of Post-Quantum Cryptography . . . 241
Lattice-Based Cryptography Primitives . . . 244
Lattices . . . 244
Computational Problems on Lattices . . . 244
Average-Case Problems on Standard Lattices . . . 245
Classes of Lattices . . . 246
Ring-LWE Based PKE Scheme . . . 247
Computationally Intensive Components of LWE (and Variants) . . . 248
Coprocessors for the Lattice-Based Cryptography . . . 254
General Optimisation Strategies . . . 254
Performance Benchmarks . . . 255
Coprocessors Design Paradigms for Lattice-Based Cryptography . . . 256
Optimization Strategies for Implementation of Underlying Components . . . 261
Physical Protection of Lattice-Based Cryptography . . . 266
Timing Attacks . . . 267
Power Analysis Attacks . . . 267
Fault Attacks . . . 268
Challenges in the Post-Quantum Cryptography Adaptation . . . 269
Conclusions . . . 270
References . . . 271

A. Khalid (✉) · D.-S. Kundi
Centre for Secure Information Technologies (CSIT), Queen's University Belfast, Belfast, UK
e-mail: a.khalid@qub.ac.uk; d.kundi@qub.ac.uk

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_8

Abstract

The impending realization of scalable Quantum computers has led to active research in Post-Quantum Cryptography. Amongst various classes of Quantum-resistant cryptographic schemes, Lattice-based cryptography is emerging as one of the most viable replacements; five out of seven 3rd round finalists in the NIST Post-Quantum Cryptography competition are Lattice-based in their construction. This chapter surveys the practicality of deployment of some of these schemes as coprocessors for high-speed applications. The study of Post-Quantum Cryptography schemes as coprocessors on custom hardware platforms is imperative since they are much more computationally intensive, compared to their classical counterparts used today. Additionally they generally have much larger key sizes, requiring higher storage budgets as well. In this context the performance requirements of a cryptosystem, area, power and memory footprints, are bench-marked, along with side channel attack resistance (if provided). This chapter discusses Lattice-based cryptosystems implemented on configurable hardware (primarily Field Programmable Gate Arrays (FPGAs) and some Application-Specific ICs results), embedded processors for light-weight IoT end-node devices, Application-Specific Instruction Set Processors (ASIPs) and a few hardware software coprocessing designs. Most of these reported results are manually optimised with a few high level synthesis optimisations. The chapter concludes by identifying some research gaps and an outlook for the future.

Keywords

Post-Quantum Cryptography (PQC) · Lattice-based cryptography (LBC) · Key Encapsulation Mechanisms (KEM) · Learning With Errors (LWE) · Module-LWE · Ring-LWE · Cryptosystem · Hardware acceleration · Crypto coprocessor

Introduction

Modern-day cryptography is based on a number of problems which are hard for classical computers to solve. This includes both of the major classes of modern cryptography, i.e., the symmetric key cryptography and the public key cryptography. The security of symmetric key cryptography algorithms relies on the hardness of the brute-force search algorithm. Examples of symmetric key cryptography are Advanced Encryption Standard (AES), Data Encryption Standard (DES) and Triple-DES (3DES). The security of public key cryptographic algorithms relies on the hardness of factoring large integer numbers and the discrete logarithm problem (including the elliptical curve discrete logarithm problem). Examples of public key cryptography used extensively include RSA, Digital Signature Algorithm (DSA), Elliptic Curve Cryptography (ECC), Elliptic Curve Digital Signature Algorithm (ECDSA), etc.

Due to the advances in the field of Quantum computing, the security of the digital world of today stands at the brink of a *complete and urgent overhaul*. This is primarily because of the following two Quantum algorithms that will affect the security of the modern-day cryptographic algorithms.

- *The Grover's Algorithm* (Grover 1996) affects only the symmetric key algorithms. By virtue of this algorithm, someone with a Quantum computer will be able to find a number in an unordered list of length N in $\sqrt{N}$ time, rather than the classical algorithm which takes $(N \div 2)$ time. Hence in a post Quantum world, the key sizes of symmetric key cryptographic algorithms should be doubled to gain half the security level, i.e. a AES-256 with a 256 bit key would provide an AES-128 equivalent security.
- *The Shor's Algorithm* (Shor 1994) enables a Quantum computer to carry out prime factorisation to be solved in polynomial time, making RSA-based cryptosystems unviable. Shor's Algorithm can also be used to solve the discrete logarithm problem, leaving the majority of modern public key cryptography vulnerable and unusable in a post-Quantum world.

The threat posed by the Quantum computers has led to active research into new cryptographic schemes which would be hard to solve on both classical and Quantum computers. These schemes are termed generally in scientific literature as *Post-Quantum Cryptography* (PQC) or *Quantum-Resistant Cryptography* (QRC) or *Quantum-safe cryptography*. Post-Quantum Cryptography will replace the public key algorithms that make the backbone of all cryptographic protocols used today. These algorithms can run on classical computers used today, yet remain secure against known Quantum computing attacks. The National Institute of Standards and Technology (NIST) initiated a Post-Quantum Cryptography (PQC) competition in 2016 to standardise Quantum resilient cryptosystems for the key exchange and digital signatures (Moody 2016). The NIST PQC competition has performed multiple rounds of evaluation of schemes submitted, with a much lower number of candidate schemes going forward to the next round. At the moment the 3rd round of the NIST-PQC is underway with seven finalists scheme and an expected date to conclude in 2022–2024 with a suite of acceptable candidates for standardisation (Moody 2016).

The various Post-Quantum Cryptography schemes under consideration in the NIST PQC come from a very diverse background of mathematical problems. What remains common for all of these Quantum-resistant algorithms is the fact that they are more complex than the currently deployed public key techniques, with much larger key sizes in comparison, making them at times, impractical for low-cost devices. This has rightly led to an active research community undertaking coprocessors for PQC on a range of platforms. This chapter provides a survey of the challenges for that and the successes achieved. In section "Post-Quantum Cryptography (PQC)", we discuss major classes of Post-Quantum Cryptography (PQC) and NIST efforts for PQC standardisation. Evidence and reasons for popularity of Lattice-based schemes over the others are discussed as well. Section "Lattice-Based Cryptography Primitives" looks at the fundamental mathematics and some

critical security-proofs of Lattice-based cryptography. Section "Coprocessors for the Lattice-Based Cryptography" highlights the implementation challenges of LBC on hardware for authentication, key sharing and digital signatures. We take up the major bottlenecks to performance acceleration and discuss the strategies for tackling them most efficiently for real-time experimentation of Post-Quantum Cryptography on custom hardware. Various scientific results in the context are categorised and bench-marked. To conclude the chapter, some gaps in the research are identified, and future directions of viable research are presented in section "Conclusions".

Post-Quantum Cryptography (PQC)

NIST Post-Quantum Cryptography Standardisation Project

The NIST Post-Quantum Cryptography Standardisation Project (referred generally as NIST PQC) was announced in 2016 with the aim of selecting and standardising post-Quantum cryptographic algorithms in the three categories of digital signature, public-key encryption and key establishment (Moody 2016). This invited the research community to submit candidate proposals on an online forum, to be publicly assessed for collaborative discussions throughout several rounds of iterations, for choosing a suite of standardisation proposals in the next few years (Fig. 1 shows the NIST PQC timeline). NIST aimed to choose a diverse range of cryptographic primitives, preferably serving as drop-in replacements, with perfect forward secrecy, resistance to side-channel attacks, simple yet flexible with inherent misuse resistance as important properties for consideration for proposal submissions to be successful. Performance on various classical platforms was also meant to be evaluated.

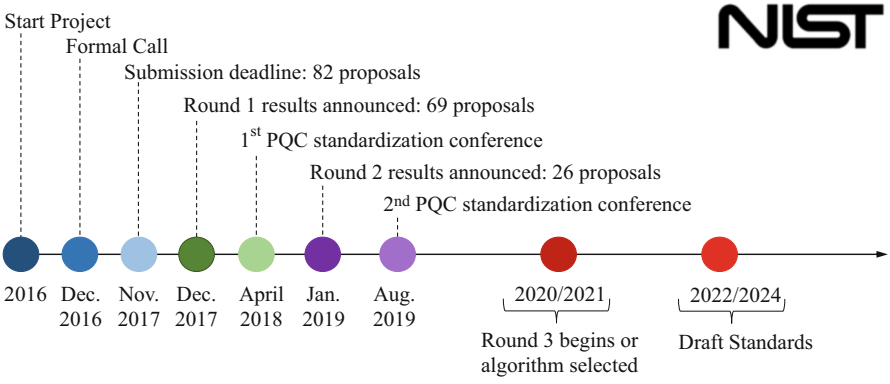


Fig. 1 NIST PQC standardisation timeline

Initial Submissions

There were a total of 82 submissions to the NIST PQC, in the form of PKE (Public Key Exchange), DSS (Digtial Signature Schemes) and KEM (Key Encapsulation Mechanism) schemes. The submissions spanned the five main Quantum-safe families, along with some miscellaneous and classical cryptography bases.

NIST's PQC Round1

A total of 69 of the original 82 submissions made it through to Round 1 following the initial inspection by NIST, which involved checking they conformed to the submission criteria. Within the first few weeks of publication, 12 of the 69 schemes were significantly attacked or broken, 5 schemes were considered beyond repair and were therefore withdrawn and others were patched. Scrutinising of the 1st round candidates took place from publication in December 2017 until January 2019. The first NIST PQC Standardisation Workshop was held in Fort Lauderdale in April 2018 and encouraged cryptanalysis and performance bench-marking of the schemes from the community as a whole.

NIST's PQC Round2

NIST used factors such as a candidate's track record in security vulnerabilities and potential feasibility of the schemes in useful applications to narrow down the selection for Round 2. They reported a lack of full confidence in the security of some schemes (NIST 2019), and others were deemed too inefficient and did not proceed to the 2nd round. NIST aimed to maintain diversity of schemes for Round 2, and to facilitate this process, encouraged teams with similar schemes to merge.

NIST's PQC Round3

On July 22, 2020, NIST announced a handful of 3rd round finalist schemes as well as some 'alternate schemes' (NIST 2020). Out of the four PKE finalist schemes, three are Lattice-based, while three of the finalist DSS schemes, two are Lattice-based. For the eight alternate schemes, NIST clarified that they are also advanced into the third round and may still potentially be standardised, although that most likely will not occur at the end of the third round (NIST 2020). It is estimated that this third phase of evaluation and review will last 12–18 months.

Classes of Post-Quantum Cryptography

In the past few years, research community has focused on several powerful cryptographic primitives that are not vulnerable to Quantum attacks. University research groups, governments worldwide, standards bodies such as the NIST and the European Telecommunications Standards Institute (ETSI) and companies like Google, IBM and many others are currently researching Post-Quantum Cryptography for this purpose. There are several strands of PQC currently being examined by the research community.

Code-Based

Code-based cryptography is based on the underlying one-way functions, i.e. error-correcting codes that are considered to be NP-hard (Wieschebrink 2006). There are two classic code-based cryptography systems named after Robert McEliece (McEliece 1978) and Harald Niederreiter (Niederreiter and Xing 2009), their inventors. For encryption of plain text, the message is encoded either by adding errors into the message or encoding a message into an error sequence; error correction during decryption recovers the original message. Code-based cryptography can provide not only the PKE, KEMs and DSS schemes, but also other cryptographic functions including identification schemes, random number generators and hash functions. A challenge surrounding code-based systems is their very large key sizes which render their implementation impossible on embedded devices with very limited resources. However, the confidence on them is boosted by their mathematical roots, e.g. McEliece scheme (and its variants) has remained secure for 40 years. The Classic McEliece scheme is part of the four PKE finalist schemes, and BIKE is included in the alternate schemes in the third round of NIST PQC (NIST 2020).

Multivariate-Based

The hardness of solving non-linear multivariate equation structures over finite fields is the foundation of Multivariate Cryptography schemes as seeking a solution for such structures is an NP-complete/-hard problem (Ding and Petzoldt 2017). They have been more successfully used to build digital signature schemes but not PKE. Multivariate signatures have the advantage of being fast and having short signature sizes; however key sizes are large and security proofs are lacking, with security estimates solely based on known attacks. Rainbow is a Multivariate-based signature scheme that made it to the finalist list of 3rd round of NIST PQC, while GeMSS (A Great Multivariate Short Signature) is included into the alternate signatures list as well (NIST 2020).

Hash-Based

The security of hash-based signatures relies on the collision resistance of the underlying hash function. Examples of hash-based schemes are Merkle (Merkle 1989) and XMSS (Buchmann et al. 2011). Their security is well understood; they offer promising performance as well as small signature sizes, making the hash-based signatures a promising Quantum-safe alternative. One of the challenges is creating practical stateless hash-based schemes. Stateful signature schemes require keeping track of the signing keys to ensure they are never reused. SPHINCS+ is an example of a practical stateless hash-based signature that is also included into the alternate list of the 3rd round of NIST PQC (NIST 2020).

Isogeny-Based

One of the most recent additions to the Quantum-resistant cryptography is based on the hardness of isogenies over elliptic curves (Jao and De Feo 2011). The Isogeny-based schemes are attractive due to their short key sizes. Further cryptanalysis

is required to increase the confidence in their security margins, as some attacks have been proposed in recent years, the isogeny-based Diffie-Hellman-type scheme CSIDH has undergone enlightening cryptanalysis in Peikert (2020). SIKE (Supersingular Isogeny Key Encapsulation) scheme is included into the alternate list of the 3rd round of NIST PQC (NIST 2020).

Lattice-Based

Arguably the most popular of the post-Quantum contenders are the Lattice-based cryptography (LBC) schemes. They are characterised by their associated worst-case hardness problems, upon which both basic such as public-key encryption (PKE) (Regev 2010), Key Exchange Mechanism (KEM) and digital signature schemes (DSS) (Ducas et al. 2013) and the more advanced cryptographic primitives can be built upon. A popular lattice problem is Learning With Errors (LWE) (Regev 2005; Daniele and Oded 2009); many LWE-based schemes have proven to be just as, if not more, efficient than existing comparable primitives.

The alternate security primitives to replace the current ones must keep up with the rapid developments in technology as a whole. For instance, with the societal shift towards the *Internet of Things*, ensuring security and privacy for an increasing number of heterogeneous connected devices is a crucial concern for which the Quantum-safe cryptographic family must cater for. In this context, the advanced cryptographic primitives by lattices offer the most adaptable and versatile security solutions. Google has trialled the Lattice-based Quantum-safe scheme NewHope in its Chrome browse in 2016 (Braithwaite 2005), and the Lattice-based DSS called BLISS-B has been integrated into the StrongSwan IPSec implementation 3 (Steffen et al. 2005).

Lattice-based cryptography schemes stand out because of the following various reasons:

- Lattice problems have been shown to have a worst-case to average-case hardness property, such as finding short vectors in a lattice (Ajtai 1996), meaning an average instance of the problem is as hard as the hardest instance.
- The Lattice cryptographic implementations are notable for their efficiency, primarily due to their inherent linear algebra-based matrix/vector operations on integers. The performance of Lattice-based primitives has been shown to be competitive with existing schemes such as RSA and ECC.
- Lattices are the most versatile and can provide PKE, DSS and KEM. Moreover, lattices are capable of more advanced cryptographic primitives too, such as Identity-Based Encryption (IBE), Attribute-Based Encryption (ABE), Fully Homomorphic Encryption, privacy-preserving encryption, Multi-party computation (MPC), e-voting, etc., which have not yet been realised by any other type of PQC cryptographic schemes.

Lattice-based schemes made the majority of NIST PQC initial submissions, with 39% of Round 1 candidates out of a total 69 being Lattice-based in construction. Lattices stayed popular later too, with 12 out of the 26 Round 2 candidates and 5

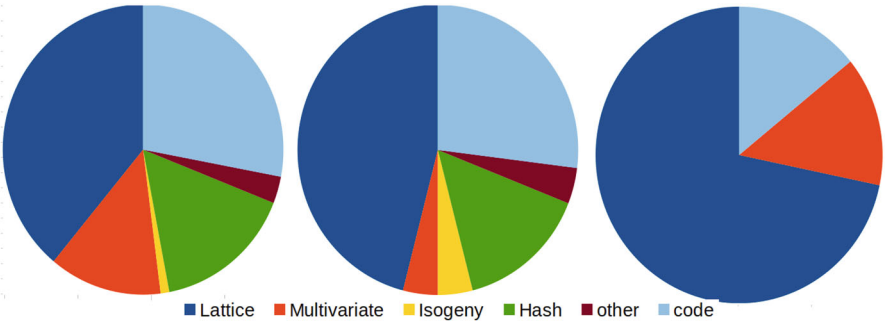


Fig. 2 NIST PQC Rounds (from left to right for Round 1, 2, 3) percentage composition of the candidates according to their types. Round 1, 2 and 3 have 69, 26 and 7 (finalists only) candidates in total with 39%, 46% and 71% candidates Lattice-based in their construction, respectively

out of 7 Round 3 candidates being Lattice-based. Figure 2 shows the percentage distribution of the candidates in each round according to their type.

Lattice-Based Cryptography Primitives

Lattices

A lattice is a mathematical structure being characterized by a regular arrangement of infinite set of points in a n-dimensional Euclidean space. Given n linearly independent vectors $\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n \in \mathbb{R}^m$, a lattice $\mathcal{L}$ can be defined as $\mathcal{L}(\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n) = \{\sum x_i \ \mathbf{b}_i \mid x_i \in \mathbb{Z}\}$, where $\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n$ represent the basis of the lattice (the definition provided in Daniele and Oded 2009). Similarly, we can defined a basis $\mathbf{B}$ as $m \times n$ matrix with $\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n$ as its column vectors such that $\mathbf{B} = \{\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n\} \in \mathbb{R}^{m \times n}$. Hence, the Lattice-based on matrix $\mathbf{B}$ will be $\mathcal{L}(\mathbf{B}) = \{\mathbf{B}x \mid x \in \mathbb{Z}^n\}$, where $\mathbf{B}x$ is the usual matrix-vector multiplication.

Computational Problems on Lattices

The **shortest vector problem (SVP)** (Ajtai 1998) and the **closest vector problem (CVP)** (Daniele 2001) are the two basic computational problems on lattices and are foundation of many LBC schemes.

- **SVP**: Given a lattice $\mathcal{L}$ generated by basis $\mathbf{B}_0$ and $\mathbf{B}_1$, find a shortest nonzero vector λ_1 inside it w.r.t $\| \mathcal{L}_2 \|$.
- **CVP**: Given a lattice $\mathcal{L}$ generated by basis $\mathbf{B}_0$ and $\mathbf{B}_1$ and a target point C (not necessarily on the lattice), find a closest point to C in lattice and associated vector λ_2 w.r.t $\| \mathcal{L}_2 \|$ as shown in Fig. 3 of 2D representation of a lattice.

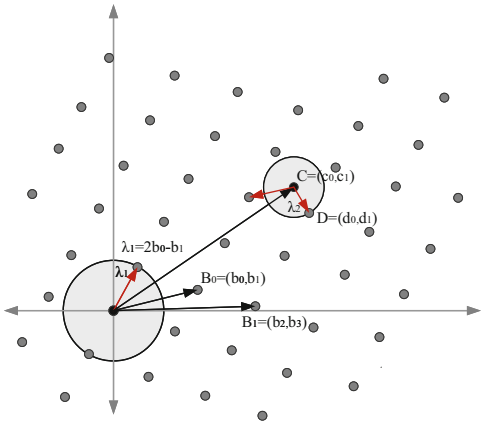


Fig. 3 Description of SVP and CVP lattice problems (Aydin et al. 2013)

The underlying lattice problems CVP and the SVP are assumed to be non-deterministic polynomial-time (NP)-hard problems, which means they cannot be solved in polynomial time (Ajtai 1996) and are resistant to Quantum computing attacks. Indeed, many schemes are based on the hardness of approximating the solution of these problems to within polynomial or super-polynomial factors. It is common for cryptographic primitives to base their security on average-case problems which have been proven to be at least as hard as the core lattice problems.

Average-Case Problems on Standard Lattices

Other than the usage of standard lattice problems like the SVP or CVP, the two most commonly used average-case problems are the **short integer solution (SIS)** (Ajtai 1996) and the **learning with errors (LWE)** (Regev 2005) problems that became the foundation of many Lattice-based schemes.

- **SIS problem**: This states that given a random matrix $\mathbf{A} \in Z_q^{m \times n}$ chosen uniformly, it is difficult to find a non-zero short vector $v \in Z_q^m \setminus \{0\}$ such that $v^T \mathbf{A} = 0^T$ and norm of $v \leq \beta$, where β represents the norm-bound, q the integer modulus.

The SIS problem was first introduced by Ajtai in (1996), and based on it the public-key cryptosystem was provided by Ajtai and Dwork in (1997). Afterwards, it served as a foundation of first practical Lattice-based cryptosystem by Hoffstein, Pipher and Silverman in 1998, i.e. the encryption scheme NTRU (Hoffstein et al. 1998). To date the encryption scheme NTRUEncrypt has withstood cryptanalytic scrutiny provided the parameters are chosen correctly, but the NTRU-based digital signature scheme NTRUSign is considered broken. However, a modified version of the signature scheme has been submitted to the NIST post-Quantum call, along with

many other proposals. Another prominent Lattice-based digital signature scheme is BLISS proposed by Ducas et al. (2013) and Ducas (2014). The security of BLISS is based on NTRU-lattices and SIS. However, although it is not a submission to the NIST standardisation initiative, it has been adopted in the strongSwan IPsec project (Steffen et al. 2005).

- **LWE problem**: This states that given a polynomial number of samples of the form $(a, \langle a, s\rangle + e)$, it is difficult to determine the secret vector $s \in Z_q^n$, where n is a power-of-two integer, defining the input dimension and q is a prime modulus, the vector a is sampled uniformly at random from Z_q^n and the error e is sampled from the appropriate error distribution.

The LWE problem was first introduced by Regev in (2005), as a generalization of the learning parity with noise (LPN) problem (Blum et al. 1994). The LWE-based cryptosystem introduced in the same work served as a foundation of several modern LBC proposals being submitted to the PQC standardisation process, i.e. PKC/KEM, NewHope, CRYSTALS-Kyber, SABER, FrodoKEM, Round5, as well as signature schemes, CRYSTALS-Dilithium, FALCON, etc. Moreover, the LWE problem is very versatile; it is used to construct many KEM/PKE schemes providing chosen-ciphertext attack (CPA) security in addition to chosen-plaintext attack (CCA) security (Regev 2010) (which was a requirement for the NIST call). CPA security means that the scheme is mathematically secure against an attacker who has access to a limited amount of plaintext/ciphertext pairs while CCA security on the other hand implies that an attacker has access to a decryption oracle as well. This security can be extended by assuming an adaptive attacker (CCA2), whereas signature schemes are designed with unforgeable under chosen-message attacks (EUF-CMA) security. This means that an attacker with access to a signing oracle is unable to forge a valid signature of a new message. Strong existential unforgeability under Chosen Message Attacks (SEUF-CMA) is an even stronger security notion that also assumes that an attacker is unable to forge a different signature of a message that he has already seen.

Classes of Lattices

There are three classes of lattices that are relevant to LBC schemes, and all these classes have in common that they require computations with large matrices that either need a lot of memory or require costly on-the-fly computations.

- **Standard or random Lattice-based schemes**: These schemes are based on standard LWE/SIS class of lattices; examples from the NIST's PQC standardisation process are the PKC/KEM, FrodoKEM, Round5 schemes, etc. But standard LBC schemes require computations with large matrices that need a lot of memory and also result in large key sizes; hence an acknowledged challenge is the large key sizes; these large-sized public keys can restrict performance. However, it is

common to add structure to the lattices to reduce this by a square root factor, at the expense of a stronger security assumption.

- **Ideal or ring Lattice-based schemes**: An alternative to standard lattices is the ideal/ring lattices while maintaining the hardness of an original problem; in this case now the security of the constructed schemes is based on ring variants of original problem, hence, the Ring-LWE and Ring-SIS. Examples from the NIST's PQC standardisation process are the PKC/KEM shemes, Newhope, NTRU and NTRU Prime while signature schemes, qTesla, Falcon, etc. In ring lattices, the matrix that is used in standard lattices is now represented by a single row, and the remaining rows are generated by cyclic shifts of the first row. Therefore ring Lattice-based schemes are more efficient as they require less memory, and the main arithmetic operation is polynomial multiplication instead of matrix-vector multiplication. As they are more efficient, the additional structure in the lattice might also be exploitable by attacks. To date, this is not known to introduce any vulnerability to the security, while allowing great benefits in efficiency. Not only are the key sizes reduced, but the speed of underlying operations can be increased due to this new structure.
- **Module Lattice-based schemes**: To have a trade-off between the efficiency of ideal/ring lattices and the trust in the security of standard lattices, **module lattices** were introduced. In other words, the module variant provides a middle ground between standard and ideal/ring lattice by reducing the algebraic structure present in ideal and increases the security without compromising much on the computational efficiency. The security of module lattices is once again based on variants of the original mathematical problems, hence, Module-LWE or Module-SIS. Examples from NIST's PQC standardisation process are PKC/KEM shemes, CRYSTALS-Kyber, SABER and Three Bears while digital signature schemes (DSS), CRYSTALS-Dilithium. In module lattices the matrix has small dimensions as defined by parameter k, and the coefficients of the matrix are no longer simple integers but entire polynomials. The value of k varies from 2, 3, 4 as in case of CRYSTALS-Kyber providing the increased security level (which is discussed in section "Performance Benchmarks") (Bos et al. 2019; Yao et al. 2021).

The current Round 3 NIST PQC candidates (Finalist/Alternatives) along with their categories of lattices are summarised in Table 1.

Ring-LWE Based PKE Scheme

The Ring-LWE (or R-LWE) problem is first proposed by Lyubashevsky et al. in (2010), while maintaining the hardness of an original problem. The R-LWE-based cryptosystems use ideal lattices, which correspond to ideals in the ring $\mathbb{Z}[x]/\langle f \rangle$, where f is an irreducible polynomial of degree n. For better efficiency and security, the ring is often taken as $R_q = \mathbb{Z}_q[x]/(x^n + 1)$ where n is a power-of-two integer representing lattice dimension and q is a prime modulus such that $q \equiv 1 \bmod 2n$.

Table 1 Round 3 NIST PQC standardisation process candidates (NIST 2020)

Schemes	Finalists/alternatives	Algorithms	Lattice class
PKE/KEM	Finalists	CRYSTALS-Kyber	Module
		NTRU	Module
		SABER	Module
	Alternatives	FrodoKEM	Standard
		NTRU Prime	Ring
DSS	Finalists	CRYSTALS-Dilithium	Module
		FALCON	Ring

The entire process of R-LWE-based PKE encryption and decryption protocol is outlined by Algorithm 1. The inputs to the algorithm are the public parameter and secret key on the basis of which public key is derived through the Key Generation step and also the Plaintext. During the encryption process as given by line 1– 3, two ciphertext were calculated, c_1 and c_2 (comprising the encoded Plaintext), while during the decryption as given by line 4–5, a plaintext was decoded after calculating the c from ciphertexts. In the algorithm, all the operations are based on polynomials on the ring $\mathbb{R}_q$, U represents the uniform distribution and D_σ is a discrete Gaussian (DG) distribution with a standard deviation σ and mean $\mu = 0$, whereas '$\times$', '$+$' represent the modular polynomial multiplication and polynomial addition operations, respectively.

Algorithm 1 The R-LWE based PKE protocol

Input: Public parameter: $a \leftarrow U$; Secret key: $r_2 \leftarrow D_\sigma$; Public key: p; Plaintext: $m \leftarrow \{0, 1\}$.
Output: Ciphertext: c_1, c_2; Decrypted plaintext: m'.
1: Sampling to generate $e_1, e_2, e_3 \leftarrow D_\sigma$;
2: Encoding plaintext $\bar{m} = encode(m)$;
3: Ciphertext is calculated $c_1 = a \times e_1 + e_2, c_2 = p \times e_1 + e_3 + \bar{m}$;
4: Decrypting $c = c_1 \times r_2 + c_2$;
5: Decoding plaintext $m' = decode(c)$;
6: **return** c_1, c_2, m'.

Computationally Intensive Components of LWE (and Variants)

From an architectural point of view, R-LWE-based PKE algorithm has two computationally intensive components: the sampling of values from a discrete Gaussian-distributed random source, i.e. D_σ, and the calculation of a series of linear algebraic operations specially modular polynomial multiplication ($\times$). The algorithmic background of these components is given below.

Discrete Gaussian Sampling

Discrete Gaussian sampling is a core component in both Lattice-based encryption and signature schemes; it provides the framework for more efficient implementations with smaller ciphertexts (Howe et al. 2018). In the implementation a number of parameters are important, the **standard deviation** σ, the **precision parameter** λ and the **tail-cut parameter** τ. σ is the most important as choosing the wrong parameter could make the problem much easier, and therefore less secure. λ decides the security level of λ-bits; it describes the distance between the perfect Gaussian distribution and one used, which should not exceed $2^{-\lambda}$. In a practical implementation τ tells us how much of the tails can be excluded.

In the context of discrete Gaussian sampling, it is important to understand that no finite computation can produce a perfect discrete normal distribution (Dwarakanath and Galbraith 2014). Consequently, distributions used are a certain degree indistinguishable from a normal distribution such that the statistical distance between two distributions have a negligible difference. This distance is a precision parameter and is typically measured in bits (i.e. $2^{-\lambda}$ for a target security of λ bits). Saarinen (2015) proved that in most cases half of the precision of discrete Gaussian samplers is sufficient as a security parameter, hence halving the size of cumulative distribution tables in both hardware and software implementations. Another important measure in the context of optimised Gaussian samplers is the Rényi divergence that enables efficient parameter choices, e.g. a smaller standard deviation (Takashima and Takayasu 2015). An algorithmic optimisation, proposed by Peikert in (2010), uses the convolution property of distributions, enabling the use of a smaller standard deviation to generate two or more samples, which have the required larger standard deviation when combined using basic additions and scalar multiplications.

In addition to the traditional method of rejection sampling, several sampling techniques have been proposed or adapted for use in Lattice-based cryptosystems: Bernoulli sampling (Ducas et al. 2013), Cumulative Distribution Table (CDT) sampling (also known as inversion sampling) (Peikert 2010), Knuth-Yao sampling (Dwarakanath and Galbraith 2014) and discrete Ziggurat sampling (Buchmann et al. 2013). For further details, and a more mathematical description on discrete Gaussian sampling within Lattice-based cryptography, the reader is referred to the review paper by Dwarakanath and Galbraith (2014).

Rejection Sampling: Rejection sampling requires using a random sample from some simple distribution and using probabilities to decide whether to reject or accept each given sample. One issue with rejection sampling is that many samples may be required before one is accepted. To overcome the high rejection rate, several of the following techniques are optimizations of basic rejection sampling. Rejection sampling has a non-constant execution time, but the timing leakage from rejection sampling is not considered to be exploitable by an attacker since the output value is independent from the number of previous rejections.

Cumulative Distribution Table (CDT) Sampler: which as the name suggests involves a pre-computed table of Gaussian cumulative distribution function values. It is a sorted table so a algorithm is used to find the desired value. There are number of ways to increase efficiency such as hashing a number of bits. When no RAM was used in Howe et al. (2018) this provided the best results in encryption.

Knuth-Yao Sampler: this is a tree based algorithm for discrete Gaussian sampling. If you have N samples which are all represented by no more than λ-bits this can be represented as a binary matrix which can be used to create a binary tree. This tree is called the discrete distribution generating tree. It contains λ levels and two types of nodes: internal nodes, which have two children and terminal nodes, which have none. To create a sample a "walk" is taken from the root node downwards, going onto the right child if the node is 1 and left if it is 0. Once a terminal node is reached it outputs the index number associated with that same. Where RAM was used in the hardware implementation in Howe et al. (2018) this method provided the best results and has a lower area than CDT.

Bernoulli Sampler: Bernoulli sampling is an optimized method of rejection sampling that incorporates the use of the Bernoulli distribution, which is a discrete distribution with only two outcomes, 0 and 1. The advantage of the Bernoulli method is that the rejection probability can be reduced compared to traditional rejection sampling.

Discrete Ziggurat Sampling: Discrete Ziggurat sampling was adapted by Buchmann et al. (2013) from an original method of continuous Ziggurat sampling as proposed by Marsaglia and Tsang (2000). It is also an optimized form of rejection sampling, which divides the area under the curve into several rectangles of equal area, where the left hand side of each rectangle is aligned with the y-axis and the right hand side of each rectangle aligns with the curve of the Gaussian distribution. This structure can then be used to optimize rejection sampling of random points from a uniform distribution. The increase in the number of rectangles then decreases the probability of rejection. One advantage of this method is that the Knuth-Yao algorithm aims at requiring, as close as possible to, the minimum number of inputs from a uniform distribution (Pöppelmann and Güneysu 2013).

Polynomial Multiplication

In order to perform decryption and encryption operations, modular polynomial multiplication must be performed which is a quite computationally intensive task. Generally, it is computed using a schoolbook-based polynomial multiplication (SPM) algorithm or a number theoretic transform (NTT).

Schoolbook Algorithm

This is the naive way to calculate the polynomial multiplication. It uses direct multiplication and accumulates the results, which can make it quite slow. Hence, when used for standard/ideal lattices, the algorithm has a quadratic complexity of $O(n^2)$.

It requires n^2 modular multiplications and $(n-1)^2$ modular additions/subtractions. However, it is versatile being applicable to any LBC parameter-set and its variant, i.e. LWE/M-LWE/R-LWE, etc., and is easier to implement at a lower hardware cost. An example for its use in R-LWE is provided in Liu et al. (2019), with both the algorithm and mathematical representation. See the mathematical representation below. The product $a(x) \times b(x)$ can be computed by (1):

$$\begin{aligned} a(x) \times b(x) &= \left[\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a_i b_j x^{i+j} \right] \bmod (x^n + 1) \\ &= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} (-1)^{\lfloor (i+j)/n \rfloor} a_i b_j x^{(i+j) \bmod n} \end{aligned} \tag{1}$$

Each coefficient is reduced modulo q and reduction modulo n is performed by using bit mask when n is power of two, whereas $(-1)^{\lfloor (i+j)/n \rfloor}$ refers to the sign bit in the accumulation, determined as follows:

$$\lfloor power \rfloor = \begin{cases} 0 & i + j < n \\ 1 & n \leq i + j \leq 2n - 2 \end{cases} \tag{2}$$

Number Theoretic Transform (NTT)

Number theoretic transform (NTT) is deemed more attractive for polynomial multiplication due to its efficient execution having quasi-linear complexity of $O(n \log n)$ (Valencia et al. 2017). The NTT is primarily a variant of the fast Fourier transformation (FFT) algorithm, which was originally used to reduce the computational complexity of discrete Fourier transform (DFT) but can also be used in NTT (Nussbaumer 1981). For Lattice-based cryptography NTT is used in polynomial multiplication, along with some modular reduction methods. It is defined over a finite field or ring R_q rather than a field of floating point/complex numbers. As a consequence, all computations carried out are within the finite field or ring R_q, with the complex roots of unity in FFT replaced with integer roots of unity. However, NTT only works with a particular set of parameters, i.e. parameters that support the circular convolution property (Nussbaumer 1981). The lattice dimension n and modulus q must be prime and satisfy $q \equiv 1 \bmod 2n$.

An example for its use in R-LWE is provided in Roy et al. (2014). For the NTT transformation, the coefficients of a polynomial $a(x) = (a_0, a_1, a_2, \ldots, a_{n-1})$ are expressed as $\hat{a}(x) = (\hat{a_0}, \hat{a_1}, \hat{a_2}, \ldots, \hat{a}_{n-1})$. More precisely, providing the coefficients; $a(x)$ and primitive n-th root of unity; ω_n also called as twiddle factors, the NTT computes the $\hat{a}(x) = \sum_{j=0}^{n-1} a_j \omega_n^{ij} \bmod q, i \in [0, n-1]$ in $O(n \log n)$. Due to the orthogonality relations between the n-th roots of unity, the inverse-NTT (INTT) can be computed simply as $a(x) = \frac{1}{n} \sum_{j=0}^{n-1} \hat{a}_j \omega_n^{-ij} \bmod q, i \in [0, n-1]$. Hence, the product of two polynomials, $a(x) \times b(x)$, can be easily calculated as $INTT(NTT(a(x)) * NTT(b(x)))$, where $*$ denotes the point-wise multiplication.

However, NTT computations are usually represented by an iterative NTT algorithm as provided in Algorithm 2; it is similar to the FFT with the NTT inner loop involving butterfly computations (i.e. line 7–8).

Algorithm 2 Iterative NTT Algorithm (Longa and Naehrig 2016)

Input: $A(x) \in \mathbb{Z}_q[x]/(x^n + 1)$
Input: n-th primitive root of unity $\omega \in \mathbb{Z}_q, n = 2^l$
Output: $\hat{A}(x) = NTT(A(x))$
1: **for** $(i = 1; i \leq l; i = i + 1)$ **do**
2: $m = 2^{l-i}$
3: **for** $(j = 0; j \leq 2^{i-1} - 1; j = j + 1)$ **do**
4: **for** $(k = 0; k \leq m - 1; m = m + 1)$ **do**
5: $U \leftarrow A[2 \cdot j \cdot m + k]$
6: $V \leftarrow A[2 \cdot j \cdot m + k + m]$
7: $A[2 \cdot j \cdot m + k] \leftarrow U + V$
8: $A[2 \cdot j \cdot m + k + m] \leftarrow (U - V) \cdot \omega^{2^{(i-1)} \cdot k}$
9: **end for**
10: **end for**
11: **end for**
12: return $\hat{A}$

Apart from the straightforward implementation of NTT and INTT, efficient algorithms are provided in Liu et al. (2017) to further enable the fast computations of NTT. The Cooley-Tukey (CT) radix-2 decimation-in-time (DIT) algorithm is the simplest implementation of the NTT that enables an $O(n \log n)$ complexity (Cooley and Tukey 1965). The algorithm recursively splits the problem into the even and the odd inputs of NTT. It requires the input polynomial values to be arranged in a bit-reversed re-ordering to enable in-place computation of the NTT. The algorithm computes the CT butterfly, calculating $c + \omega^N d$ and $c - \omega^N d$ for $N \in \{0, n/2 - 1\}$ and $\omega, c, d \in Z_q$. This fast algorithm can be used to compute both the NTT and the inverse NTT (INTT). The powers of the primitive roots of unity can either be computed on-the-fly or pre-computed and stored to save the resources for the on-the-fly computations (exponentiations), and the trade-off is ROM vs Multipliers. The CT-DIT algorithm requires a bit-reversal step before starting the NTT computation as the algorithm takes bit-reversed inputs and produces naturally ordered output. Bit-reversal is required again before INTT computation if it is using the CT-DIT computation butterfly as well. To avoid the bit reversed transformations of the arrays within the design, the NTT computation is carried out using the CT-DIT butterfly, and for INTT, the computations are carried out using the decimation in frequency (DIF) technique, called gentleman-sande (GS) (Longa and Naehrig 2016) butterfly. This DIF NTT algorithm also recursively splits the computation into even and odd sub-problems and expects the input array in bit reversed order.

The R-LWE-based PKE protocol utilising the frequency domain NTT is different from the ordinary R-LWE scheme (provided by Algorithm 1). The Algorithm 3

outlines the protocol utilising NTT technique. Firstly, we need to transform each input into the NTT domain as given by line 4 and then perform the computations as required by the R-LWE scheme. And for the final decrypted message, we need to apply the INTT to convert it back to the normal domain as given by line 7.

Algorithm 3 R-LWE based PKE using NTT

Input: Public parameter: $\hat{a}$ Public key: $\hat{p}$; Secret key: $\hat{r_2}$; Plaintext: $m \leftarrow \{0, 1\}$.
Output: Ciphertext: $\hat{c}_1, \hat{c}_2$; Decrypted plaintext: m'.
1: Sampling to generate $e_1, e_2, e_3 \leftarrow D_\sigma$;
2: Encoding plaintext $\bar{m} = encode(m)$;
3: Computing $e_3 m = e_3 + \bar{m}$
4: Computing $\hat{e_1}$ = NTT(e_1), $\hat{e_2}$ = NTT(e_2), $\hat{e_3 m}$ = NTT($e_3 m$);
5: Ciphertext is calculated $\hat{c_1} = \hat{a} * \hat{e_1} + \hat{e_2}$, $\hat{c_2} = \hat{p} * \hat{e_1} + \hat{e_3 m}$;
6: Decrypting $\hat{c} = \hat{c_1} * \hat{r_2} + \hat{c_2}$;
7: Computing $c = \text{INTT}(\hat{c})$
8: Decoding plaintext $m' = decode(c)$;
9: **return** c_1, c_2, m'.

Barrett's Reduction

This is not a method of performing the multiplication in itself but a method of improving performance when doing the modular portion of the multiplication. It was first presented in 1986 after digital signal processors (DSP) were introduced (Barrett 1986). Barrett's reduction replaces the costly divisions to get the modulo result with multiplications and a subtraction as given by Algorithm 4. It includes an estimation of how often the modulus has to be subtracted to obtain a result smaller than the modulus. To accelerate the Barrett's modular reduction in hardware, mostly

Algorithm 4 Barrett's Reduction Algorithm

Input: x: unsigned number, q = prime modulus
Output: y: x mod q
1: $t = \lfloor \frac{x}{q} \rfloor$;
2: $tq = t \times q$;
3: $y = x - tq$;
4: **while** $(y \geq q)$ **do**
5: $\quad y = y - q$;
6: **end while**
7: **return** (y);

a technique of shift-addition-multiplication-subtraction-subtraction (SAMS2) as proposed in Liu et al. (2015) is adopted in the implementations (Fan et al. 2018; Kundi et al. 2020; Zhang et al. 2020).

Coprocessors for the Lattice-Based Cryptography

This section summarises various high-speed implementation strategies for Lattice-based cryptography. In this context, appropriate performance benchmarks are first elaborated, as they vary for implementation platform. The optimization strategies of Lattice-based designs are discussed in general and of various constituent ingredient implementations of the LBC algorithms are discussed in particular. The state-of-the-art LBC implementations on various implementation platforms are compared in the end.

For the ubiquitous IoT devices, embedded portable processors are preferred over the configurable hardware platforms. These embedded processors have limited computational power and I/O capabilities and consequently have some additional constraints on which algorithms can be mapped on them. Due to their limited power supply, energy-efficient implementations with lower security levels are preferred. Additionally, their limited internal memory resources only let the PKE/KEM schemes with ciphertext/ encapsulated key that can easily be accommodated. For digital signatures too, a small-sized public key, small digital signature, and a range of supported hash output sizes are recommended.

Other than these generic optimization goals, the design engineer may consider the platform-dependent optimisation strategies whenever possible. The system components may include General Purpose Processors (GPPs), Graphics Processing Units (GPUs), Application-Specific Integrated Circuits (ASICs) and Field Programmable Gate Arrays (FPGAs), all of which may require different expertise and optimisation strategies. Modern Graphics Processor Units (GPUs) offer a many-core architecture on which parallel homogeneous threads are executed in a Single Instruction Multiple Thread (SIMT) fashion. Symmetric cryptographic algorithms have been reported to achieve remarkable performance speedups, with much fewer results for public key cryptography. A cryptosystem developer should have the necessary expertise to juggle up often conflicting design requirements and constraints. Often the design cycle is repetitive and tedious as the designer must explore a huge design space of microarchitectural optimisations intelligently before reaching a desired solution.

General Optimisation Strategies

Optimisation of an algorithm always has a specific goal in mind, e.g. to better fit the constraints imposed by a target platform (set the word size of the architecture as bit size of the modulus, etc.). The optimisation goal is consequently dependent on the choice of an appropriate target platform of implementation with specific features. Lattice-based cryptographic algorithms generally are more *data-flow centric* instead of being *control-flow centric* and are consequently better suited for hardware optimisations since a simple control flow leads to a simple state machine and a better higher utilisation of the underlying configurable hardware components. Similarly,

cryptographic algorithms with multiple instances of a module that run in parallel can be optimised easily on custom hardware, e.g. XOR operations, multipliers on DSPs, etc.

Performance Benchmarks

To enable a fair performance evaluation, it is critical to identify the right performance benchmarks (or optimisation goals), as they may vary for different applications. For constrained devices a better energy-efficiency (and lower memory stack usage) can easily outweigh a higher throughput. Performance critical applications instead aggressively exploit the parallellisation possibilities to meet the high throughput requirements at the cost of higher resource utilisation. The developer generally juggles between various optimisation goals to pick a point in the design space best suited for an application. Some general efficiency considerations of LBC coprocessor implementations are listed below.

- **Security Strength**: For most of the LBC algorithms, the choice of security strength dictates a trade-off between performance (cost, resource, latency) and the required application security level. In the NIST call for PQC, the proposals invited had to classify the range of their algorithms security strength equivalent to the existing NIST standards in symmetric cryptography, i.e. (in order of increasing strength) a security strength of 1,. . . ,5 implies that any brute force cryptanalytic effort requires computational resources comparable to (or greater than those) required for key search on AES block cipher or finding a hash collision on SHA-2 (Moody 2016), as shown in Table 2. The security levels 4 and 5 are considered for high-security use-cases only, e.g. government, military, long-term data storage, etc. For IoT applications, higher security levels are generally less desirable due to their associated resource overheads.
- **Low Resource Usage**: For configurable hardware platforms, reduction in the area occupied generally directly translates to a less expensive and more energy-efficient design. A typical technique to do that is serial execution of operations sharing the same components. If block RAM units are used, they could be stored to maximum capacity to keep the number of used memory components small even though that means that only one memory access per clock cycle is possible (two for dual-port memories). Similarly, pre-computed tables can be calculated

Table 2 The security levels specified by NIST in their call for submissions (Moody 2016)

Security level	At least as difficult to break as. . .
1	AES-128
2	SHA256
3	AES-192
4	SHA384
5	AES-256

on-the-fly to avoid storing them in memory blocks. A low-power implementation can also be achieved by running the device at a lower clock frequency.

- **Throughput Performance**: For real-time applications, a low latency (high throughput) performance is crucial. On custom hardware platforms (including FPGAs and ASICs) performance-oriented design strategies are *parallelism* and *Pipelining* that usually mean the reverse of minimization of the design resource usage. To allow faster memory access, multiple block RAM instances can be used to allow multiple read accesses in one clock cycle on FPGAs. Digital signal processors (DSP) can be used to speed up arithmetic operations. Pipelining increases the operating frequency of a design by reducing its critical path. On the downside, it results in an increase of the number of clock cycles to finish the operation and higher processing elements cost. Loops in which one iteration step does not depend on the previous one can be unrolled to be executed in parallel and therefore increase the performance.
- **Side Channel Attacks Resistance**: For security-critical applications, side-channel attack resistance has to be inculcated in the design. An attacker with physical access to the target platform should not able to extract secret information by using side-channel information, like timing, power consumption or electromagnetic emanation (EM). The countermeasures generally incur a penalty in terms of increased use of resource. Further discussion is carried out in section "Physical Protection of Lattice-Based Cryptography".

Coprocessors Design Paradigms for Lattice-Based Cryptography

This section discusses various design paradigms and limitations of coprocessors that are special-purpose in nature to optimise the execution of Lattice-based cryptography algorithms. Based on a range of security applications such as storage devices, embedded systems, network routers, etc., these coprocessors vary in their capabilities and limitations. They perform public key encryption, public key exchange and authentication. We list below major categories and some prominent work reported in the context.

1. **Embedded IoT processors:** PQM4 is the most comprehensive open source benchmarking and testing framework for post-Quantum cryptographic schemes implemented on ARM Cortex processors (PQM4 2018). The library targets the STM32F4 Discovery board featuring an ARM Cortex-M4 CPU, 1 MB of Flash and 192 KB of RAM. This library currently contains implementations of 10 post-Quantum KEM schemes; all except one of these implementations are Lattice-based in their constructions. Three post-Quantum signature schemes have also been reported, all Lattice-based in their construction.

   Khalid et al. in (2019) survey the LBC implementations on the constrained devices (embedded microprocessors). Some favourite LBC schemes in terms of various IoT critical performance bench-marks (low-power footprint, small

area, compact bandwidth requirements and high performance) are provided. Table 3 shows some most competitive reported results of Lattice-based KEM and signature PQC schemes from the 3rd round of NIST competition. For the schemes undertaken, the middle range NIST equivalent security (level 3) is being chosen, unless otherwise specified by a suffix (−3, −5). Out of the various Lattice-based post-Quantum key encapsulation schemes, Saber stands out both in terms of its resource-constrained nature for a small memory foot print but also in terms of throughput speed performance. In Karmakar et al. (2018), the authors claim to exploit DSP instructions and efficient memory accesses for a fast implementation of polynomial multiplication. A memory efficient Karatsuba and just-in-time strategy for generating the public matrix of the module lattice is used to reduce the memory footprint; consequently *speed efficient* and *memory efficient* versions are reported for speed-memory trade-offs as shown in Table 3. The speed-optimised implementation of Saber is faster than Frodo and Kyber-3 (marginally slower in decapsulation) (PQM4 2018). Frodo is much slower than Kyber/NewHope since they are based on module/ideal lattices that are inherently more efficient as the main operation in their implementation is polynomial multiplication (Howe et al. 2018).

For signatures, the large Falcon tree used in the fast Fourier sampling in the signature generation of Falcon is the major bottleneck for memory usage and the authors of Oder et al. (2019) tried to reduce the memory footprint by merging the tree generation and the fast Fourier sampling step into a single algorithm. This results in a compact implementation; the performance for the level-1 and level-2 equivalent security levels is shown in Table 3. For CRYSTALS-Dilithium, the NTT of the reference implementation is optimised at assembly level by merging two of the eight stages of the NTT to reduce memory accesses (Güneysu et al. 2018). CRYSTALS-Dilithium takes the lead here in terms of better overall throughput performance compared to both qTESLA and Falcon while qTESLA reference implementation from PQM4 (2018) has smaller stack requirements. Reference to classical schemes is also given for comparison.

2. **Application-Specific Hardware with Little Flexibility:** High speed Lattice-based cryptosystems have only been proposed for reconfigurable devices primarily on FPGAs and very few on ASICs. The choice of FPGAs is justified since the Quantum-resistant schemes are still undergoing scrutiny of the cryptographic community before their standardisation and the ease of re-programmability offered by FPGAs is therefore attractive. ASIC fabrication generally requires mass production to be profitable. Many optimisations proposed for the FPGA implementation of LBC can also be considered for ASIC designs; a naive direct mapping into an ASIC design is not likely. In this context, some opportunities and challenges are identified by Oder et al. (2016). ASIC implementations of Lattice-based schemes can offer much better power efficiency and design compactness. Furthermore, well-known power optimisation techniques such as power gating and clock gating are used for better energy efficiency. Unlike FPGAs in which the LUTs seldom reach full utilisation, logic used in an ASIC has much smaller granularity (single gates) enabling a much

Table 3 Dynamic memory usage (in bytes) and the Clock cycle counts for various leading Lattice-based PQC NIST 2nd round KEM contestants on an ARM Cortex-M at 168 MHz

Scheme	Ref.	Operation	Cycles	Time (ms)	Stack (Bytes)
Lattice-based PQC KEMs					
Saber (speed)	Karmakar et al. (2018)	Key Gen	1,147,000	7	13,883
		Enc.	1,444,000	9	16,667
		Dec.	1,543,000	9	17,763
Saber (memory)	Karmakar et al. (2018)	Key Gen	1,165,000	7	6931
		Enc.	1,530,000	9	7019
		Dec.	1,635,000	10	8115
Kyber-1	PQM4 (2018)	Key Gen	726,921	4	6456
		Enc.	987,864	6	9120
		Dec.	1,018,946	6	9928
Kyber-3	PQM4 (2018)	Key Gen	1,200,291	7	10,544
		Enc.	1,446,284	9	13,720
		Dec.	1,477,365	9	14,880
Kyber-5	PQM4 (2018)	Key Gen	1,771,729	11	15,664
		Enc.	2,142,912	13	19,352
		Dec.	2,188,917	13	20,864
FrodoKEM -AES	Howe et al. (2018)	Key Gen	101,273,066	603	35,484
		Enc.	106,933,956	637	63,484
		Dec.	107,393,295	639	63,628
FrodoKEM -cSHAKE	Howe et al. (2018)	Key Gen	187,070,653	1114	33,800
		Enc.	253,735,550	1510	57,968
		Dec.	254,194,895	1513	58,112
Lattice-based PQC signatures					
Falcon-1	Oder et al. (2019)	Key Gen.	114,546,135	682	63,652
		Sign	80,503,242	479	63,653
		Verify	530,900	3	63,654
Falcon-5	Oder et al. (2019)	Key Gen.	365,950,978	2178	120,596
		sign	165,800,855	987	120,597
		verify	1,046,700	6	120,598
CRYSTALS Dilithium	Güneysu et al. (2018)	Key Gen.	2,320,362	14	50,488
		Sign	8,348,349	50	86,568
		Verify	2,342,191	14	54,800
Classical schemes					
ECC-256	UM0586 (2018)	Key Gen.	12,713,277	76	–
		Sign	13,102,239	78	–
		Verify	24,702,099	147	–
RSA-2048	UM0586 (2018)	Key Gen.	–	–	–
		Sign	228,068,226	1358	–
		Verify	61,951,481	369	–

more compact design. Similarly the routing in ASICs entertains only the specific logic used, unlike a generic routing components for multiple FPGA elements.

For lightweight IoT end-node devices, high security levels are less desirable due to their associated overhead; consequently level 3 security or close is used; generally a minimum of 112-bits is employed (McKay et al. 2016). The lightweight hardware implementation of symmetric block ciphers requires 2– 4K Gate Equivalents (GEs) with reasonable performance while asymmetric cryptography (ECC processor) involves 10k–12k GEs (having respective security level of 113-bit to 131-bit) (Eisenbarth et al. 2007) that now need to be replaced by the LBC algorithms having resistance to the Quantum attacks. The complexity of LBC calls for much more complex designs with a higher budget.

The state-of-the-art reported ASICs based LBC implementation proposed are summarised in Table 4. In (2018), Song et al. designed Lattice Encryption Instruction Accelerator (LIEA) chip for R-LWE scheme, employing NTT-based multiplier and Discrete Gaussian sampler with SHA-256 for seed generation. The chip consumed very low energy (119 nJ) for parameter for 106-bit security with 776k GE that is significantly high for passive IoT devices. In another study (Oder et al. 2019; Nejatollahi et al. 2018) using *gem5-aladdin* to explore the design space of LBC schemes for the domain-specific accelerators ranging from resource-constraint IoT devices to high-performance computing applications. The exploration workflow provides an early estimate of area and power consumption for various parameter sets targeting FPGA as well as ASICs platforms. The minimum achievable gate count for NTT-based multiplier module (NTT_small) on ASIC using 45 nm technology 189k GE at a cost of considerably high energy consumption is 596.86 nJ. Basu et al.

Table 4 ASICs based state-of-the-art LBC implementations

Ref.	Tech. (nm)	Parameters (n,q,σ)	Freq. (MHz)	Module mm^2/KGE[a]	Area (mW)	Power	Energy (nJ)
Song et al. (2018)	40	(256,7681,4.51)	300	NTT, Sampler, SHA-256	2.05/776	160.1	119
Nejatollahi et al. (2018)	45	(512,12289,–)	100	NTT_fast, NTT_small,	0.89/567	35.60	1016
Basu et al. (2019)	65	(512,12289,2.83) (512,76881,1)	169 200	NTT, Sampler, SHA-3	3.2K/1273 3.3K/1341	38.02 39.21	69420 6210
Fritzmann and Sepúlveda (2019)	65	(256,7681,–) (512, 12289,–) (1024,12289,–)	25	NTT,	0.32K/131	6.168 5.976 5.926	352 765 1684
Banerjee et al. (2019)	40	(512,12289,2.83) (512,7681,1)	72	NTT, Sampler SHA-3	0.28/106	7.79 8.33	5790 7740

[a]GE Gate Equivalent in terms of NAND2

in (2019) used High Level Synthesis (HLS)-based design space exploration to benchmark the estimated resources for the ASICs platform. Later, Fritzmann and Sepúlveda (2019) presented the ASICs-based design of NTT module for the LBC supporting three parameters set on a 65 nm technology (UM0586 2018). The design employed clock gating and operand isolation techniques to reduce the power consumption and achieved comparative energy consumption to a 45 nm technology with an additional capability to cater for Side-Channel Attacks (SCA) because of susceptibility of NTT-based polynomial multiplication to SCA attacks. Lastly, Banerjee et al. (2019) presented Sapphire cryptographic chip using on TSMC 40 nm low-power CMOS technology with NTT-based multiplier. The chip occupies a 106k GE with 40.25 kB SRAM for a 128-bit security level.

As referred in section "Computationally Intensive Components of LWE (and Variants)", the bottleneck in Lattice-based cryptography constituent operation is the calculation of linear algebraic multiplication. In general, the performance of this polynomial multiplication unit dictates the performance of the whole cryptosystem. Additionally, the discrete error sampling (Gaussian/Uniform distributions) is also critical since a naive implementation based on rejection sampling could be too slow to be practical and too leaky not to be vulnerable to side channel attacks. These two critical blocks are taken up further in section "Optimization Strategies for Implementation of Underlying Components" for FPGA implementations. The ASIC implementations are already reported in Table 4.

3. **Hardware/Software Co-Design:** Hardware-software co-design approach lets a scheme run on a processor while letting the coprocessor hardware execute the computationally intensive parts of design. It combines the fast software development with the acceleration advantage of hardware implementations. In Fritzmann et al. (2019) a RISC-V based Lattice-based cryptography is presented with NTT transformation and hash generation chosen for acceleration. In Jati et al. (2019) an ARM processor is coupled with an FPGA fabric to accelerate the compute-intensive modules to achieve remarkable performance results.
4. **Application-Specific Instruction Set Processors (ASIP):** An ASIP is an application-specific instruction-set processor with a dedicated design for a particular application, with its instruction set specifically designed to accelerate heavy and most used functions of that application. ASIP architecture is designed to implement the assembly instruction set with minimum hardware cost. Sapphire (Banerjee et al. 2019) extends a RISC-V processor core with application-specific instruction set extensions to support different Module-LWE/Ring-LWE operations and their CCA/CPA-secure implementations, i.e. the error sampling and polynomial computations. The challenge of ASIP designer is to reach the highest performance over silicon, over power consumption as well as over the design cost on the face of all the pipeline design overhead as compared to a specific purpose designed hardware.
5. **High-Level Synthesis:** In one of recent study (Basu et al. 2019), Basu et al. used High-Level Synthesis (HLS)-based design space exploration and benchmark the estimated resources for the various LBC schemes targeting FPGAs and ASICs platforms. But the resource utilizations in terms of GE/power/energy are

worse than the implementation of dedicated ASICs/FPGA-based coprocessors. Similarly, Nejatollahi et al. in (2018) provide a workflow using gem5-aladdin to explore the design space of LBC schemes for the domain-specific accelerators ranging from resource-constraint IoT devices to high-performance computing applications. The exploration workflow provides an early estimate of area and power consumption for various parameter sets targeting FPGA as well as ASICs platforms.

Optimization Strategies for Implementation of Underlying Components

As referred in section "Computationally Intensive Components of LWE (and Variants)", the most prominent Lattice-based cryptography constituent operations are the calculation of several linear algebraic operations such as addition/subtraction, division/multiplication and the sampling of values from a discrete random source (Gaussian/Uniform distributions). Schemes that are based on standard lattices, like the LWE, usually require matrix-vector multiplication, while the ring variants of these problems, like R-LWE, require polynomial multiplication often carried out using the traditional multiplication methods: schoolbook or NTT (satisfying $q \equiv 1 \mod 2n$). Signature schemes usually require sampling from a Gaussian distribution with a higher standard deviation than encryption schemes. There are also schemes which feature only uniform sampling during signing and verification, and Gaussian sampling is only performed during key generation, e.g. TESLA (Alkim et al. 2015).

We now consider these aspects to define building blocks that represent key components for each hardware implementation.

Discrete Gaussian Sampling

In the context of discrete Gaussian sampling implementations, there are two reasons for an increased interest in relevant research activity. *Firstly*, rejection sampling techniques, if not optimized, can severely hamper the performance of the system. *Secondly*, the discrete Gaussian samplers performing in non-constant time can be exploited through timing attacks.

1. **Sampling by Inversion** Inversion sampling uses stored, pre-computed Cumulative distribution tables (CDT) which are then searched to generate suitable discrete Gaussian samples (DuG and Bai 2015). A hardware design of CDT, targeting the Xilinx Spartan-6 FPGA platform, was proposed by Pöppelmann et al., and compared with the Bernoulli approach, both schemes were incorporated into hardware designs of the BLISS signature scheme (Pöppelmann et al. 2014). CDT outperforms Bernoulli in terms of both operations per second and in area usage. Several optimisations are proposed, such as an optimised binary search using shortcut tables, reduction of table sizes by using the Kullback-Leibler divergence

and, as previously mentioned, the use of the convolution property to reduce the size of the pre-computed tables. In order to implement the CDT approach in constant time, Güneysu (Pöppelmann and Güneysu 2013) proposed the input value to be compared to all table entries using parallel comparators. Howe et al. (2018) instead recommended using binary search algorithm to reach the required value in the CDT table, resulting in a very lightweight and yet a constant-time alternative, outperforming it by a factor of around 5× in terms of Ops/s/S (operations per second per slice) compared to the one proposed in Pöppelmann and Güneysu (2013).

2. **Bernoulli Sampling:** Bernoulli sampling is optimised rejection sampling using the Bernoulli distribution. In (2014), Oder et al. compared Bernoulli, Knuth-Yao and Ziggurat Gaussian sampling methods on a Cortex-M4F microcontroller concluding that the Bernoulli performs better than Knuth-Yao and Ziggurat, in terms of both memory requirements and speed. Bernoulli sampler is leaky in terms of timing, and to achieve constant time, it has to (Howe et al. 2018) perform many more comparison than the designs previously proposed in the literature.
3. **Knuth-Yao Sampling:** In (2013), Roy et al. proposed a hardware design of Knuth-Yao sampling for use in LWE encryption schemes. The proposed optimisations include the implementation of a discrete distribution generating tree, optimised storage of a probability matrix and also the column-wise compression of zeros existing in the probability matrix. The Knuth-Yao random walk does not operate in constant time and was slow due to bit by bit scanning. In Howe et al. (2018) a pre-calculation was suggested improving the Ops/s/S of the Knuth-Yao Sampling by a factor of 2. Both of these implementations were non-constant time and therefore vulnerable to timing attacks. Roy et al. (2014) proposed random shuffling of samples to counter timing attacks.
4. **Discrete Ziggurat Sampling:** Buchmann et al. (2013) proposed a C++ implementation with several optimisations and compared Ziggurat with alternative sampling methods to show that Ziggurat is suitable when large standard deviations are required. Discrete Ziggurat sampling has a non-constant running time. The only hardware implementation (that was constant time as well) was proposed in Howe et al. (2018) showed that despite requiring a large amount of area resources, the discrete Ziggurat sampler offers generally a higher throughput.
5. **Binomial Sampling:** More recently, most of the Lattice-based cryptographic schemes requiring narrow Gaussian distributions use binomial distribution-based sampling techniques. A reduction using the Rényi divergence (Takashima and Takayasu 2015) is provided that shows that the usage of a binomial distribution instead of a rounded continuous Gaussian distribution will not provide an attacker with a significant advantage. In order to sample from a binomial distribution, one has to count the Hamming weight of two random bit vectors. For small standard deviations (commonly used in Lattice-based encryption schemes but not in signature schemes), this approach leads to very efficient samplers that do not require any pre-computed tables and run in constant time. However, for larger standard deviations, the binomial sampler suffers from long running times since the length of the bit strings increases as the square of the standard deviation.

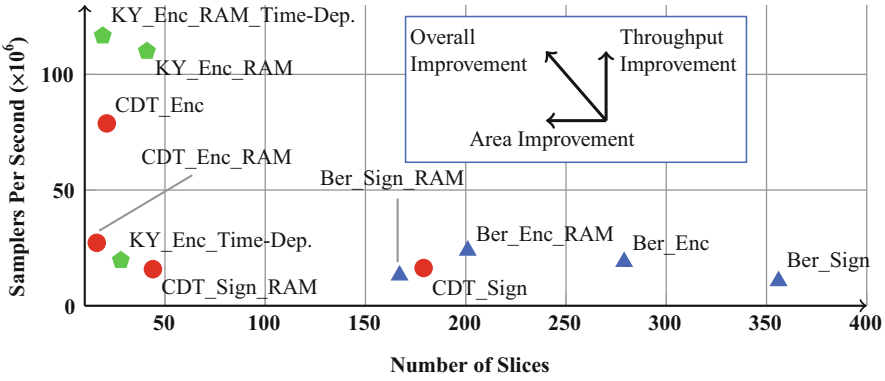


Fig. 4 Graphical performance results of all proposed discrete Gaussian samplers, on the Spartan-6 LX25-3 FPGA, with and without RAM use. All results are time-independent unless otherwise stated (Time-Dep.) (Howe et al. 2018)

In Howe et al. (2018), thorough investigation of all the practical discrete Gaussian samplers (CDT, Knuth-Yao, Bernoulli and discrete Ziggurat) used in Lattice-based cryptosystems is carried out. Novel time-independent implementations are presented, ensuring resistance against timing attacks; the designs also focus on low area foot-print and high throughput. A survey of all FPGA-based implementations reported to date is presented and analysed with the proposed hardware sampler designs clearly outperforming most of the previously proposed samplers. Figure 4 plots the post-PAR results for CDT, Knuth-Yao (KY), Bernoulli (Ber) and discrete Ziggurat (DZigg) samplers, both with and without the use of RAMs, targeted to the same FPGA device. For encryption, where a design does not use any RAM, the discrete Ziggurat sampler (DZigg_Enc) is the fastest, offering approximately 97 operations per second, but with a large area requirement. However, the RAM-free CDT (CDT_Enc) sampler surpasses all others in terms of an overall balanced performance with area, throughput and timing independence. If the use of additional BRAMs is considered, the Knuth-Yao time-independent implementation (KY_Enc_RAM) has the best overall performance in terms of low area, high throughput and also the lowest number of bits required per sample. For signatures, the RAM-free CDT implementation (CDT_Sign) proves to be an overall winner, followed by the discrete Ziggurat sampler (DZigg_Sign), and the Bernoulli sampler (Ber_Sign), being around 2x more expensive in terms of slices, which also boasts a better throughput per slice.

Polynomial Multiplication

In the context of polynomial multiplication, selecting either standard (LWE), module (M-LWE) or ideal (R-LWE) lattices in Lattice-based constructions has the significant impact on the system in terms of efficiency. Therefore, optimised strategies for polynomial multiplication have been one of the core interests by

the research community. For hardware designs, the modulus-q dictates the size of the multiplier or multipliers (i.e. $\lceil \log_2 q \rceil$-bit) required in a hardware design of a cryptosystem using any class of lattices. It is to be noted that hardware designs of Lattice-based cryptosystems using LWE or M-LWE rather than R-LWE cannot be considered low-area due to the inherently large key sizes required in such cryptosystems. However, matrix-vector multiplication is a traditional digital signal processing (DSP) task, and there are dedicated and optimised DSP hardware units available on FPGAs that can be targeted to accelerate the performance of these operations. This could improve the efficiency of LWE/M-LWE hardware designs requiring matrix-vector computations. Moreover, the number of parallel multipliers used in a design determines the area-throughput trade-off for various applications ranging from lightweight to high-speed networks.

1. **Schoolbook Multiplication** The schoolbook algorithm is very versatile and is applicable to any class of the lattices with any modulus-q, but the choice of schoolbook algorithm can have a detrimental effect on a systems efficiency due to two aspects: firstly, the high storage requirements since a matrix (LWE/M-LWE)/vector (R-LWE) must be stored and secondly, due to large computational requirements of matrix-vector/polynomial multiplications with quadratic complexity O(n^2). Table 5 lists the schoolbook multiplication accelerators for the LBC having resource consumption (in terms of LUTs/FFs/Slices, DSPs, BRAMs) and performance results (in terms of frequency, number of cycles, execution time).

   Mostly, the schoolbook multiplication is targeted for the lightweight applications (Fan et al. 2018; Zhang et al. 2020; Pöppelmann and Güneysu 2014; Liu et al. 2019), with very few implementations targeting high-speed network applications (Roy and Basso 2020; Dang et al. 2019). The naive schoolbook multiplication-based designs proposed in Fan et al. (2018) and Pöppelmann and Güneysu (2014) employed the R-LWE scheme. Both the designs resulted in a very high latency, i.e. greater than $2n^2$ cycles and for $n = 256$, required 1.07 ms (Pöppelmann and Güneysu 2014) to 456 µs (Fan et al. 2018) for the complete encryption phase only, whereas the optimised R-LWE architecture proposed in Liu et al. (2019) and Zhang et al. (2020) (utilising the signed Gaussian data and accommodating 2 multiplications per DSP) focused to improve the overall latency of the schoolbook multiplication design in Fan et al. (2018), reducing the encryption time to 229 µs (gian 2× performance) and 129 µs (4× improved performance), respectively. The schoolbook multiplication-based high-speed designs proposed in Roy and Basso (2020) and Dang et al. (2019) support the M-LWR LBC scheme (i.e. Saber). The architecture with 256 parallel multiply-and-accumulate (MAC) units for $n = 256$ and matrix_dimension $l = 2$ (i.e. 2× computations of R-LWE) takes maximum of 41 µs (Dang et al. 2019) while with 512 parallel MAC units takes 26.7 µs (Roy and Basso 2020) for the complete decapsulation phase. Increasing further the matrix dimension will have little effect on the overall latency as the next data block will be processed in the pipeline, once it is filled (Dang et al. 2019).

Table 5 Schoolbook-based multiplication accelerators

Implementation	Device	Type	LUTs/FFs/Slices	DSPs	BRAMs	Freq. (MHz)	# of Cycles	Exec. Time (μs)
R-LWE Fan et al. (2018)	K-7	Enc	1098/407/337	1	0	288	131,604	456
		/Dec	609/318/182	1	0	288	/65,802	/228
R-LWE Pöppelmann and Güneysu (2014)	S-6	Enc	360/290/114	1	2	128	136,986	1.07k
		Dec	162/136/51	1	1	179	66,304	370
R-LWE Liu et al. (2019)	K-7	Enc	898/815/303	1	3	304	69,654	229
		Dec	635/190/194	1	1	303	34,436	114
R-LWE Zhang et al. (2020)	K-7	Enc /Dec	1381/1179/479	2	2	275	35,478 /17,732	129 /64.5
M-LWR Roy and Basso (2020) (LightSaber)	Ul+	Encap /Decap	24.9k/10.7k /≈4.2k[a]	0	2	150	3135 /4012	20.9 /26.7
M-LWR Dang et al. (2019) (LightSaber)	Ul+	Encap /Decap	12.3k/11.2k/2k	256	3.5	322	13,846 /13,202	43 /41

[a]Estimates while comparing LUTs with implementations on same FPGA device; UltraScale+ (Ul+) has 8 LUTs per Slice as compared to Spartan-6 (S-6) and Kintex-7 (K-7)

2. **NTT multiplication**: For the polynomial multiplication, NTT (in hardware) is also extensively studied; a complete survey is provided in Valencia et al. (2017). It takes $O(n \log n)$ operations instead of $O(n^2)$ in $\mathbb{Z}_q[\mathbf{x}]$ using the convolution property of NTT; however the modulus-q must satisfy $q \equiv 1 \bmod 2n$. Table 6 lists some of the NTT multiplication accelerators for the LBC having resource consumption (in terms of LUTs/FFs/Slices, DSPs, BRAMs) and performance results (in terms of frequency, number of cycles, execution time).

   The NTT-based most compact designs resorted to a single butterfly unit (Pöppelmann and Güneysu 2013; Roy et al. 2014; Liu et al. 2018) while others targeting high-performance, employed multiple butterfly units, higher radixes, i.e. radix-4 and radix-8, and pipelined approaches, i.e. multi-path delay commutator (MDC), etc. (Rentería-Mejía and Velasco-Medina 2017; Du et al. 2016; Feng et al. 2020; Chen et al. 2020; Huang et al. 2020). The compact NTT designs in Pöppelmann and Güneysu (2013) and Roy et al. (2014) targeting the R-LWE scheme with $n = 256$, achieved 26 and 20 μs per encryption, respectively, while Liu et al. (2018) takes 4.5 ms. The high-speed NTT-based R-LWE designs reduced the computational time many-fold, and a pipelined radix-2 MDC (R2MDC) based architecture resulted in a 5.08 μs (Rentería-Mejía and Velasco-Medina 2017) for encryption process while with a four

Table 6 NTT based multiplication accelerators

Implementation	Device	Type	LUTs/FFs/Slices	DSPs	BRAMs	Freq. (MHz)	# of Cycles	Exec. Time (μs)
R-LWE Pöppelmann and Güneysu (2013)	V-6	Enc /Dec	4549/3624/1506	1	12	262	6861 /4404	26 /16.8
R-LWE Roy et al. (2014)	V-6	Enc /Dec	1349/860/410	1	2	313	6300 /2800	20 /9
R-LWE Liu et al. (2018)	S-6	Enc /Dec	1307/889/406	0	2.5	80	360k /72k	4.5k /0.9k
R-LWE Rentería-Mejía and Velasco-Medina (2017)	St-4	Enc	28,105/28,358/–	26	220[a]	232	1194	5.08
(R2MDC)		Dec	4122/6178/–	22	15[a]	237	644	4.32
R-LWE Du et al. (2016)	S-6	Enc	–/–/953	16	4.5	246	917	3.72
NTT Feng et al. (2020)	S-6	Enc /Dec	–/–/14,000	128	1	235	440 /220	1.87 /0.94
M-LWE Chen et al. (2020) Kyber512	A-7	Enc /Dec	442/237/≈147	1	3″	136	7197	52.92
M-LWE Huang et al. (2020) Kyber512	A-7	Enc	80,322/– /141,825	54	200.5	155	1834	11.8

[a]M9K in Startix (St) FPGA ≈ 8K BRAM, ″ one 36K BRAM = two 18K BRAMs

radix-2 butterfly units, took 3.72 μs (Du et al. 2016). In one of the recent work Feng et al. (2020), a fully parallel multi-lane radix-2 NTT algorithm based on the Stockham NTT for $n = 256$ achieved 0.94 μs for just a polynomial multiplication, whereas a pipelined NTT architecture reported in Chen et al. (2020) and Huang et al. (2020), targeting the M-LWE scheme, i.e. CRYSTALS-Kyber with matrix_dimension $l = 2$, attained 52.92 and 11.8 μs, respectively, for the polynomial multiplication part only.

Physical Protection of Lattice-Based Cryptography

Physical attacks against Lattice-based constructions are a research direction largely unexplored. This is mainly due to the fact that Lattice-based constructions themselves are relatively new. A comprehensive analysis of their resistance against

physical attacks is of utmost importance before their widespread deployment since the side channel attacks protection via countermeasures may incur a massive overhead in implementation resource and/or performance latency of a system. Schemes incurring a minimal overhead side channel security are going to be preferred in NIST standardisation process (Moody 2016). A survey of side channel attacks and countermeasures for PQC schemes is available here (Khalid et al. 2018). The most common physical attacks (timing attacks, power analysis attacks and fault attacks) to Lattice-based constructions and their countermeasures are summarised below.

Timing Attacks

Timing attacks extract secret information by exploiting the differences in time required by a device to perform specific operations. Ensuring that an implementation has a constant execution timing profile prevents an attacker from deducing information about the secret key by observing the running time of the device, often at a higher cost. Several constant time implementations of discrete Gaussian distribution used in R-LWE have been undertaken. Micciancio and Walter (2017) proposed to allegorically combine several Gaussian distribution samples with small standard deviations to reach a higher one. A similar approach is followed in Pöppelmann et al. (2014). The protection of base sampler can be done via bit-slicing, as proposed by authors in Karmakar et al. (2018). Bit-slicing helps them achieve a fully constant-time sampling as well as for a substantial speedup. Constant-time hardware architectures for a wide range of samplers have been proposed by Howe et al. (2016) and Khalid et al. (2016) for standard deviations that are typical for lattice-based signature schemes and lattice-based KEMs. Roy et al. (2014) suggested the use of sample shuffling technique to avoid the expensive constant-time Gaussian sampler. Later this shuffling was attacked by Pessl (2016).

The KEM proposals in third PQC round substitute the Gaussian sampler by a binomial sampler. The binomial sampling is not vulnerable to timing attacks, and can be approximated to be used instead of an exact discrete Gaussian distribution for encryption and key exchange schemes as the noise sample precision required there is low.

Power Analysis Attacks

Power analysis attacks extract secret information by analysing correlations between the power leakage of a target device and the secret values processed during the algorithm execution. The typical countermeasures against power and EM side-channels are masking and hiding of secret data. These countermeasures generally have to be designed and implemented on top of the basic implementations and usually incur a massive overhead in implementation resource. Two popular countermeasures are Hiding and Masking.

- Hiding de-correlates the data processed by the device and its power foot print. One way of doing that is ensuing constant power consumption, or to shuffle up the order of instructions execution as undertaken in Roy et al. (2014). For ASICs or FPGA based implementation, noise generation on device is easily done, making the attackers job hard to read power traces for attacks.
- Masking is a provable countermeasure in which the sensitive variable is processed in several equal shares, and each portion is individually computed so that the attacker cannot steal the secret value unless all the shares are known. First order masking splits the secret value into two shares. First order masking on R-LWE-based schemes leads to lowered performance and security (Reparaz et al. 2015; Oder et al. 2018). In Reparaz et al. (2015) first order masking is applied on the secret key and the decoder, delaying decoding up to 16 times and increasing the decryption error probability by 19%. Oder et al. (2018) presented a masked CCA-secure R-LWE scheme with masked binomial sampler and a masked encoder; Table 7 shows its performance on an ARM Cortex-M4 microcontroller. With major overhead (71%) coming from binomial sampling, the overall performance overhead factor of the masking scheme is 5.7×.

Fault Attacks

For a fault attack the adversary purposely induces a fault and exploits the erroneous behaviour of the circuit to gain information about the secret values in the cryptosystem. A simplistic remedy is the *concurrent error detection (CED)*, generally undertaken via duplication of hardware or re-computation on the same hardware, making it resource expensive or having lowered performance, respectively.

Numerous results have been reported in the context of fault attacks and countermeasures on lattice based signatures: Bindel et al. (2016) investigated multiple lattice-based signature schemes including BLISS, ring-TESLA and GLP signatures for randomisation, skipping and zeroing faults and found several vulnerabilities in all of them. Ravi et al. (2019) presented a first order skipping fault attack

Table 7 Cycle count on ARM Cortex-M4F of R-LWE (Oder et al. 2018).

Operation	Unmasked	Masked
Key Generation	2,669,559	–
CCA2-secured Encryption	4,176,684	–
CCA2-secured Decryption	4,416,918	25,334,493
CPA-RLWE Encryption	3,910,871	19,315,432
CPA-RLWE Decryption	163,887	550,038
`SHAKE-128`	87,738	201,997
NTT	83,906	–
INTT	104,010	–
Uniform Sampling	60,014	–
Binomial Sampling	1,142,448	6,031,463
PRNG (64 bytes)	88,778	202,454

on two deterministic lattice-based signature schemes CRYSTALS Dilithium and qTESLA. They proposed to move crucial operations into the NTT domain leading to a fault affecting all coefficients of a polynomial and therefore increasing the attack complexity. McCarthy et al. (2019) presented a fault attack called BEARZ (Basis Extraction by Aborting Recursion or Zeroing) attacking the FALCON signature scheme's trapdoor sampler.

In (2019), Howe et al. analyse possible fault attacks on Gaussian and binomial samplers on hardware platforms. The proposed countermeasures check statistical properties of the distribution to detect fault errors.

Challenges in the Post-Quantum Cryptography Adaptation

Standardising and deploying a new cryptographic system is a slow and tedious process. It is mention-worthy to highlight the timeline of SHA-1 replacement in this context. SHA-1 specifications were published first in 1993; due to potential vulnerabilities spotted early on, its replacement SHA-2 was published in 2002, SHA-3 in 2015. Since 2005, SHA-1 has been considered insecure against well-funded opponents (Schneier 2005). Major web browsers however stopped using SHA-1 no earlier than 2017, and Microsoft Windows announced a cease of SHA-1 use for software signing updates in 2020 (Microsoft 2020). Since the timeline of availability of a practical quantum computer is not known with certainty, it is crucial to start planning for the migration of classical public-key algorithms to PQC algorithms now. Without that, all the public-key algorithms based cryptosystems and the associated protocols like IPsec, TLS, SSH, etc. will be vulnerable.

Understandably, major standardising bodies around the globe including NIST and ETSI are actively pursuing potential replacement of security algorithms and have released various white papers to identify the associated challenges regarding the adaption of PQC algorithms (Barker et al. 2021a; Standards 2015). Recently, the National Cybersecurity Center of Excellence (NCCoE), a part of NIST, initiated a project to highlight the issues involved in the migration from current public-key algorithms to PQC algorithms and to develop the description for their practical demonstration (Barker et al. 2021b).

Some critical hurdles/approaches for PQC adaption into existing cryptosystems are as follows (Barker et al. 2021a,b; Standards 2015):

- Most PQC schemes have larger key/signature sizes rendering larger associated memory footprints/stack usages. This may prohibit the PQC schemes to be used as a drop-in replacement of the currently used algorithms. Most of the widely used protocols like X.509, IKE, TLS, SSH, etc. are designed with the **Cryptographic Agility** in mind, i.e. have the ability to accommodate larger key/signature size extensions. Protocols lacking this ability may require fundamental changes in the messaging and data structure to withstand the quantum threats. Additionally, a larger transmission bandwidth is also required for PQC schemes.

- The PQC schemes require much more computational complexity compared to what the world is using today. Based on the choice of PQC class, this may render higher latency and comparable throughput performance (if not lower) to the public key schemes used today. Consequently, high performance applications may require much more computational power.

In the context of PQC migration, the use of **Hybrid approaches** employing both the Quantum-safe and classical public-key algorithms are advocated. IETF draft (Stebila 2020) and technical reports (Crockett et al. 2019; Xu et al. 2020) provide the "Framework" to integrate the PQC with the current ECC/RSA into TLS, SSH protocols to provide potentially secure KEM/KEP. This hybrid approach will help maintain the inter-operability during the migration.

In addition to above, the term **Hybrid approach** is also used to combine the Quantum and non-quantum schemes. Here, the Quantum scheme refers to the **Quantum key distribution (QKD)** that is based on Quantum mechanics. It enables Alice and Bob to generate shared symmetric keys securely where the source of security underpinned comes from the communication of quantum light signals. The non-quantum schemes refer to the post-quantum cryptography (PQC) schemes that run on today's (classical) computing devices and are based on mathematical techniques immune to Shor's algorithm attack and hence are deemed resistant to other quantum algorithms that may be developed in the future. A combination of PQC and QKD provides a solution very appealing in terms of security. If Alice and Bob seed a QKD session with new, asymmetric PQC, the quantum keys they establish will remain secure even if there are catastrophic failures due to advances in quantum computing, implementation vulnerabilities or evolving classical security failures (Dowling et al. 2020).

Conclusions

Lattice-based cryptography shows substantial promise as a Quantum-safe alternative to existing public-key cryptosystems. They easily become the best fit in terms of key sizes compactness and simplicity of implementation, when compared against hash based, code-based schemes. However, compared to the traditional public key schemes, the performance of LBC schemes suffers with associated large public key sizes, which is a challenge for real-world systems. This work surveys the state-of-the-art PQC coprocessor implementations on a range of applications (including FPGAs and embedded microprocessors) to give an idea how much is being achieved already. Most of the PQC implementation have proved to be at-par, if not more efficient than existing comparable primitive implementations. Their large key sizes, however, require a bigger memory footprint, large transmission bandwidths that may posed a great challenge to deploy it into the existing infrastructure, which is also under investigation.

References

Ajtai M (1996) Generating hard instances of lattice problems. In: Proceedings of 28th annual ACM symposium on theory of computing, pp 99–108

Ajtai (1998) Worst-case complexity, worst-case complexity and lattice problems. Doc Math J DMV III ICM:421–428

Ajtai M, Dwork C (1997) A public-key cryptosystem with worst-case/average-case equivalence In: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing (STOC '97).Association for Computing Machinery. New York, pp 284–293. https://doi.org/10.1145/258533.258604

Alkim E, Bindel N, Buchmann JA, Dagdelen Ö, Schwabe P (2015) Tesla: tightly-secure efficient signatures from standard lattices. IACR Cryptology ePrint Archive, vol 2015, p 755

Aydin A, Cameron P, Patrick S (2013) Low-cost and area-efficient FPGA implementations of lattice-based cryptography. In: 2013 IEEE international symposium on hardware-oriented security and trust (HOST), pp 81–86

Banerjee U, Ukyab TS, Chandrakasan AP (2019) Sapphire: a configurable crypto-processor for post-quantum lattice-based protocols. IACR Trans Cryptogr Hardw Embed Syst 2019:17–61

Barker W, Polk W, Souppaya M (2021a) Getting ready for post-quantum cryptography: exploring challenges associated with adopting and using post-quantum cryptographic algorithms, NIST Cybersecurity White Paper, p 10. [Online]. Available: https://doi.org/10.6028/NIST.CSWP.04282021

Barker W, Consulting D, Souppaya M (2021b) Migration to post-quantum crytpography, NCCoE Draft Project. [Online]. Available: https://www.nccoe.nist.gov/sites/default/files/library/project-descriptions/pqc-migration-project-description-draft.pdf

Barrett P (1986) Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In: Advances in cryptology – CRYPTO'86. Proceedings, Santa Barbara. Springer, Berlin/Heidelberg, pp 311–323. [Online]. Available: https://doi.org/10.1007/3-540-47721-7_24

Basu K, Soni D, Nabeel M, Karri R (2019) A resource-efficient and side-channel secure hardware implementation of Ring-LWE cryptographic processor. In: NIST post-quantum cryptography – a hardware evaluation study. Cryptology ePrint Archive, vol 2019. [Online]. Available: https://eprint.iacr.org/2019/047

Bindel N, Buchmann J, Krämer J (2016) Lattice-based signature schemes and their sensitivity to fault attacks. In: 2016 workshop on fault diagnosis and tolerance in cryptography (FDTC). IEEE, pp 63–77

Blum A, Furst M, Kearns M, Lipton RJ (1994) Cryptographic primitives based on hard learning problems. In: Stinson DR (ed) Advances in cryptology – CRYPTO'93. Springer, Berlin/Heidelberg, pp 278–291

Bos J, Ducas L, Kiltz E, Lepoint T, Lyubashevsky V, Schanckk JM, Schwabe P, Seile G, Stehl D (2019) CRYSTALS-Kyber: a CCA-secure module-lattice-based kem. Technical report, National Institute of Standards and Technology. [Online]. Available: https://pq-crystals.org/kyber/data/kyber-specification-round2.pdf

Braithwaite M (2005) Experimenting with post-quantum cryptography. https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html

Buchmann J, Dahmen E, Hülsing A (2011) XMSS-a practical forward secure signature scheme based on minimal security assumptions. In: International workshop on post-quantum cryptography. Springer, pp 117–129

Buchmann JA, Cabarcas D, Göpfert F, Hülsing A, Weiden P (2013) Discrete ziggurat: a time-memory trade-off for sampling from a Gaussian distribution over the integers. In: SAC 2013, pp 402–417. [Online]. Available: https://doi.org/10.1007/978-3-662-43414-7_20

Chen Z, Ma Y, Chen T, Lin J, Jing J (2020) Towards efficient Kyber on FPGAs: a processor for vector of polynomials. In: 25th Asia and South Pacific design automation conference (ASP-DAC), pp 247–252

Cooley JW, Tukey JW (1965) An algorithm for the machine calculation of complex fourier series. Math Comput 19:297–301

Crockett E, Paquin C, Stebila D (2019) Prototyping post-quantum and hybrid key exchange and authentication in TLS and SSH. Cryptology ePrint Archive, Report 2019/858. https://eprint.iacr.org/2019/858

Dang VB, Farahmand F, Andrzejczak M, Gaj K (2019) Implementing and benchmarking three lattice-based post-quantum cryptography algorithms using software/hardware co-design. In: 2019 international conference on field-programmable technology (ICFPT), pp 206–214

Daniele M (2001) The hardness of the closest vector problem with preprocessing. IEEE Trans Inf Theory 47(3):1212–1215

Daniele M, Oded R (2009) Lattice-based cryptography. In: Bernstein D, Buchmann J (eds) Post-quantum cryptography. Springer, Berlin/Heidelberg

Ding J, Petzoldt A (2017) Current state of multivariate cryptography. IEEE Secur Priv **15**(4):28–36

Dowling B, Hansen TB, Paterson KG (2020) Many a mickle makes a muckle: a framework for provably quantum-secure hybrid key exchange. In: International conference on post-quantum cryptography. Springer, pp 483–502

Ducas L (2014) Accelerating BLISS: the geometry of ternary polynomials. IACR cryptology ePrint archive, vol 2014, p 874. [Online]. Available: http://eprint.iacr.org/2014/874

Ducas L, Durmus A, Lepoint T, Lyubashevsky V (2013) Lattice signatures and bimodal Gaussians. In: Proceedings of annual cryptology conference – CRYPTO 2013. Springer, Berlin/Heidelberg, pp 40–56

Du C, Bai G, Wu X (2016) High-speed polynomial multiplier architecture for ring-lwe based public key cryptosystems. In: 2016 international Great Lakes symposium on VLSI (GLSVLSI). IEEE, pp 9–14

Du C, Bai G (2015) Towards efficient discrete Gaussian sampling for lattice-based cryptography. In: 2015 25th international conference on field programmable logic and applications (FPL). IEEE, pp 1–6

Dwarakanath NC, Galbraith SD (2014) Sampling from discrete Gaussians for lattice-based cryptography on a constrained device. Appl Algebra Eng Commun Comput 25:159–180

Eisenbarth T, Kumar S, Paar C, Poschmann A, Uhsadel L (2007) A survey of lightweight-cryptography implementations. IEEE Des Test Comput 24(6):522–533

Fan S, Liu W, Howe J, Khalid A, O'Neill M (2018) Lightweight hardware implementation of R-LWE lattice-based cryptography. In: Proceedings of IEEE Asia Pacific conference on circuits and systems (APCCAS), pp 403–406

Feng X, Li S, Xu S (2020) R-LWE-oriented high-speed polynomial multiplier utilizing multi-lane stockham ntt algorithm. IEEE Trans Circuits Syst II: Express Briefs 67:556–559

Fritzmann T, Sepúlveda J (2019) Efficient and flexible low-power ntt for lattice-based cryptography. In: 2019 IEEE international symposium on hardware oriented security and trust (HOST). IEEE, pp 141–150

Fritzmann T, Sharif U, Müller-Gritschneder D, Reinbrecht C, Schlichtmann U, Sepulveda J (2019) Towards reliable and secure post-quantum coprocessors based on RISC-V. In: 2019 design, automation test in Europe conference exhibition (DATE), pp 1148–1153

Grover LK (1996) A fast quantum mechanical algorithm for database search. In: Proceedings of 28th annual ACM symposium on theory of computing. STOC'96. ACM, New York, pp 212–219. [Online]. Available: https://doi.org/10.1145/237814.237866

Güneysu T, Krausz M, Oder T, Speith J (2018) Evaluation of lattice-based signature schemes in embedded systems. In: 25th IEEE international conference on electronics, circuits and systems (ICECS). IEEE, pp 385–388

Hoffstein J, Pipher J, Silverman JH (1998) NTRU: a ring-based public key cryptosystem. In: Algorithmic number theory. Springer, Berlin/London, pp 267–288

Howe J, Khalid A, Rafferty C, Regazzoni F, O'Neill M (2016) On practical discrete Gaussian samplers for lattice-based cryptography. IEEE Trans Comput 67(3):322–334

Howe J, Khalid A, Rafferty C, Regazzoni F, O'Neill M (2018) On practical discrete Gaussian samplers for lattice-based cryptography. IEEE Trans Comput 67(3):322–334

Howe J, Oder T, Krausz M, Güneysu T (2018) Standard lattice-based key encapsulation on embedded devices. IACR Trans Cryptogr Hardw Embed Syst 2018:372–393

Howe J, Khalid A, Martinoli M, Regazzoni F, Oswald E (2019) Fault attack countermeasures for error samplers in lattice-based cryptography. In: 2019 IEEE international symposium on circuits and systems (ISCAS). IEEE, pp 1–5

Huang Y, Huang M, Lei Z, Wu J (2020) A pure hardware implementation of CRYSTALS-Kyber PQC algorithm through resource reuse. In: IEICE electronics express, vol advpub

Jao D, De Feo L (2011) Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In: International workshop on post-quantum cryptography. Springer, pp 19–34

Jati A, Gupta N, Chattopadhyay A, Sanadhya SK (2019) Side-channel protected post-quantum cryptoprocessor. Cryptology ePrint Archive: Report, vol 2019, p 765

Karmakar A, Mera JMB, Roy SS, Verbauwhede I (2018) Saber on ARM. IACR Trans Cryptogr Hardw Embed Syst 2018:243–266

Karmakar A, Roy SS, Reparaz O, Vercauteren F, Verbauwhede I (2018) Constant-time discrete gaussian sampling. IEEE Trans Comput 67(11):1561–1571

Khalid A, Howe J, Rafferty C, O'Neill M (2016) Time-independent discrete gaussian sampling for post-quantum cryptography. In: 2016 international conference on field-programmable technology (FPT). IEEE, pp 241–244

Khalid A, Oder T, Valencia F, O'Neill M, Güneysu T, Regazzoni F (2018) Physical protection of lattice-based cryptography: challenges and solutions. In: Proceedings of the 2018 on Great Lakes symposium on VLSI, pp 365–370

Khalid A, McCarthy S, O'Neill M, Liu W (2019) Lattice-based cryptography for IoT in a quantum world: are we ready? In: 2019 IEEE 8th international workshop on advances in sensors and interfaces (IWASI). IEEE, pp 194–199

Kundi D-S, Bian S, Khalid A, Wang C, O'Neill M, Liu W (2020) AxMM: area and power efficient approximate modular multiplier for R-LWE cryptosystem. In: Proceedings of IEEE international symposium on circuits and systems (ISCAS), pp 1–5

Liu Z, Seo H, Roy SS, Großschädl J, Kim H, Verbauwhede I (2015) Efficient Ring-LWE encryption on 8-bit AVR processors. In: Proceedings of international workshop on cryptographic hardware and embedded systems, pp 663–682

Liu Z, Pöppelmann T, Oder T, Seo H, Roy SS, Güneysu T, Großschädl J, Kim H, Verbauwhede I (2017) High-performance ideal lattice-based cryptography on 8-bit AVR microcontrollers. ACM Trans Embed Comput Syst 16(4):1–24

Liu D, Zhang C, Lin H, Chen Y, Zhang M (2018) A resource-efficient and side-channel secure hardware implementation of Ring-LWE cryptographic processor. In: IEEE transactions on circuits and systems I: regular papers, pp 1–10

Liu W, Fan S, Khalid A, Rafferty C, O'Neill M (2019) Optimized schoolbook polynomial multiplication for compact lattice-based cryptography on fpga. IEEE Trans Very Large Scale Integr (VLSI) Syst 27(10):2459–2463

Liu W, Fan S, Khalid A, Rafferty C, O'Neill M (2019) Optimized schoolbook polynomial multiplication for compact lattice-based cryptography on FPGA. IEEE Trans Very Large Scale Integr (VLSI) Syst 27(10):2459–2463

Longa P, Naehrig M (2016) Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In: International conference on cryptology and network security. Springer, pp 124–139

Lyubashevsky V, Peikert C, Regev O (2010) On ideal lattices and learning with errors over rings. In: Gilbert H (ed) Proceedings of advances in cryptology – EUROCRYPT 2010. Springer, Berlin/Heidelberg, pp 1–23

Marsaglia G, Tsang WW (2000) The Ziggurat method for generating random variables. J Stat Softw 5(1):1–7. [Online]. Available: https://www.jstatsoft.org/index.php/jss/article/view/v005i08

McCarthy S, Howe J, Smyth N, Brannigan S, O'Neill M (2019) Bearz attack falcon: implementation attacks with countermeasures on the falcon signature scheme. IACR Cryptol. ePrint Archive vol 2019, p 478

McEliece RJ (1978) A public-key cryptosystem based on algebraic. In: Coding Thv, vol 4244, pp 114–116
McKay K, Bassham L, Sönmez Turan M, Mouha N (2016) Report on lightweight cryptography. National Institute of Standards and Technology, Technical Report
Merkle RC (1989) A certified digital signature. In: Conference on the theory and application of cryptology. Springer, pp 218–238
Micciancio D, Walter M (2017) Gaussian sampling over the integers: efficient, generic, constant-time. In: Annual international cryptology conference. Springer, pp 455–485
Microsoft (2020) SHA-1 windows content to be retired August 3, 2020, Windows IT Pro Blog. [Online]. Available: https://techcommunity.microsoft.com/t5/windows-it-pro-blog/sha-1-windows-content-to-be-retired-august-3-2020/ba-p/1544373
Moody D (2016) Post-quantum cryptography: NIST's plan for the future. In: Talk given at PQCrypto 16 conference, Fukuoka. [Online]. Available: https://pqcrypto2016.jp/data/pqc2016_nist_announcement.pdf
Nejatollahi H, Dutt ND, Banerjee I, Cammarota R (2018) Domain-specific accelerators for ideal lattice-based public key protocols. IACR Cryptol. ePrint Archive, vol 2018, p 608
Niederreiter H, Xing C (2009) Algebraic geometry in coding theory and cryptography. Princeton University Press, Princeton
NIST (2019) Status report on the first round of the NIST Post-Quantum Cryptography Standardization Process. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/ir/2019/NIST.IR.8240.pdf
NIST (2020) Status report on the second round of the NIST Post-Quantum Cryptography Standardization Process. [Online]. Available: https://csrc.nist.gov/publications/detail/nistir/8309/final
Nussbaumer HJ (1981) The fast fourier transform. In: Fast fourier transform and convolution algorithms. Springer, Berlin/Heidelberg/New York pp 80–111
Oder T, Pöppelmann T, Güneysu T (2014) Beyond ECDSA and RSA: lattice-based digital signatures on constrained devices. In: 2014 51st ACM/EDAC/IEEE design automation conference (DAC). IEEE, pp 1–6
Oder T, Güneysu T, Valencia F, Khalid A, O'Neill M, Regazzoni F (2016) Lattice-based cryptography: from reconfigurable hardware to ASIC. In: 2016 international symposium on integrated circuits (ISIC). IEEE, pp 1–4
Oder T, Schneider T, Pöppelmann T, Güneysu T (2018) Practical CCA2-secure and masked ring-LWE implementation. IACR Trans Cryptogr Hardw Embed Syst 2018:142–174
Oder T, Speith J, Höltgen K, Güneysu T (2019) Towards practical microcontroller implementation of the signature scheme Falcon. In: International conference on post quantum cryptography. Springer, pp 1–17
Peikert C (2010) An efficient and parallel Gaussian sampler for lattices. In: International cryptology conference – CRYPTO 2010. CRYPTO'10, Santa Barbara. Springer, Berlin/Heidelberg
Peikert C (2020) He gives C-sieves on the CSIDH. In: Annual international conference on the theory and applications of cryptographic techniques. Springer, pp 463–492
Pessl P (2016) Analyzing the shuffling side-channel countermeasure for lattice-based signatures. In: International conference on cryptology in India. Springer, pp 153–170
Pöppelmann T, Güneysu T (2013) Towards practical lattice-based public-key encryption on reconfigurable hardware. In: Proceedings of international conference on selected areas in cryptography, pp 68–85
Pöppelmann T, Güneysu T (2014) Area optimization of lightweight lattice-based encryption on reconfigurable hardware. In: Proceedings of IEEE international symposium on circuits and systems (ISCAS), pp 2796–2799
Pöppelmann T, Ducas L, Güneysu T (2014) Enhanced lattice-based signatures on reconfigurable hardware. In: International workshop on cryptographic hardware and embedded systems. Springer, pp 353–370
PQM4 (2018) Post-quantum cryptography on ARM Cortex-M4 family of microcontrollers. [Online]. Available: https://github.com/mupq/pqm4
Ravi P, Jhanwar MP, Howe J, Chattopadhyay A, Bhasin S (2019) Exploiting determinism in lattice-based signatures: practical fault attacks on pqm4 implementations of nist candidates. In: Proceedings of the 2019 ACM Asia conference on computer and communications security, pp 427–440

Regev O (2005) On lattices, learning with errors, random linear codes, and cryptography. In: Proceedings of 37th annual ACM symposium on theory of computing (STOC), pp 84–93

Regev O (2010) The learning with errors problem (invited survey). In: Proceedings of of 25th annual IEEE conference on computational complexity, CCC 2010, Cambridge, MA, pp 191–204

Rentería-Mejía CP, Velasco-Medina J (2017) High-throughput Ring-LWE cryptoprocessors. IEEE Trans VLSI Syst 25:2332–2345

Reparaz O, Roy SS, Vercauteren F, Verbauwhede I (2015) A masked ring-lwe implementation. In: International workshop on cryptographic hardware and embedded systems. Springer, pp 683–702

Roy SS, Basso A (2020) High-speed instruction-set coprocessor for lattice-based key encapsulation mechanism: saber in hardware. In: TCHES, vol 4

Roy SS, Vercauteren F, Verbauwhede I (2013) High precision discrete Gaussian sampling on FPGAs. In: International conference on selected areas in cryptography. Springer, pp 383–401

Roy SS, Vercauteren F, Mentens N, Chen DD, Verbauwhede I (2014) Compact Ring-LWE cryptoprocessor. In: Proceedings of international workshop on cryptographic hardware and embedded systems, pp 371–391

Roy SS, Reparaz O, Vercauteren F, Verbauwhede I (2014) Compact and side channel secure discrete Gaussian sampling. ePrint Report 2014/591. https://eprint.iacr.org/2014/591

Roy SS, Reparaz O, Vercauteren F, Verbauwhede I (2014) Compact and side channel secure discrete gaussian sampling. In: IACR Cryptology ePrint Archive vol 2014, p 591

Saarinen M-JO (2015) Gaussian sampling precision in lattice cryptography. Technical Report 953. [Online]. Available: https://eprint.iacr.org/2015/953

Schneier B (2005) Cryptanalysis of SHA-1, Schneier on Security. [Online]. Available: https://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html

Shor PW (1994) Algorithms for quantum computation: discrete logarithms and factoring. In: Proceedings of 35th annual symposium on foundations of computer science, pp 124–134

Song S, Tang W, Chen T, Zhang Z (2018) LEIA: a 2.05 mm 2 140 mw lattice encryption instruction accelerator in 40 nm CMOS. In: 2018 IEEE custom integrated circuits conference (CICC). IEEE, pp 1–4

Standards EWC (2015) Quantum safe cryptography and security: an introduction, benefits, enablers and challenges, White Paper No. 8

Stebila D (2020) Hybrid key exchange in TLS 1.3, Internet Engineering Task Force (IETF) draft

Steffen A, Willi M, Brunner T (2005) Strongswan IPSec project. http://www.strongswan.org

Takashima K, Takayasu A (2015) Tighter security for efficient lattice cryptography via the Renyi divergence of optimized orders. In: Au M-H, Miyaji A (eds) Provable security. Lecture notes in computer science. Springer International Publishing, pp 412–431. [Online]. Available: https://doi.org/10.1007/978-3-319-26059-4_23

UM0586 (2018) STM32 cryptographic library. [Online]. Available: https://www.st.com/resource/en/user_manual/cd00208802.pdf

Valencia F, Khalid A, O'Sullivan E, Regazzoni F (2017) The design space of the number theoretic transform: a survey. In: Proceedings of international conference on embedded computer systems: architectures, modeling, and simulation (SAMOS), pp 273–277

Wieschebrink C (2006) Two NP-complete problems in coding theory with an application in code based cryptography. In: Proceedings of IEEE international symposium on information theory. IEEE, pp 1733–1737

Xu J, Gao Y, Lim H (2020) Practical quantum-safe stateful hybrid key exchange protocol. Cryptology ePrint Archive, Report 2020/763. https://eprint.iacr.org/2020/763

Yao K, Kundi D-S, Wang C, O'Neill M, Liu W (2021) Towards CRYSTALS-Kyber: a M-LWE cryptoprocessor with area-time trade-off. In: Proceedings of IEEE international symposium on circuits and systems (ISCAS), pp 1–5

Zhang Y, Wang C, Kundi D-S, Khalid A, O'Neill M, Liu W (2020) An efficient and parallel R-LWE cryptoprocessor. IEEE Trans Circuits Syst II: Express Briefs 67(5):886–890

Fault Tolerant Architectures

9

Siva Satyendra Sahoo, Anup Das, and Akash Kumar

Contents

Introduction . . . 278
Faults, Errors, and Failures . . . 281
Fault Model . . . 281
Fault Mechanisms . . . 282
Fault Masking . . . 284
Reliability . . . 286
Fault Tolerance . . . 289
Fault Tolerance Activities . . . 290
Redundancy . . . 291
Fault-Tolerant Computation . . . 294
Single-Core Computing . . . 294
Multicore Computing . . . 296
Reconfigurable Computing . . . 297
Fault-Tolerant Memory/Storage . . . 298
Cache/On-chip SRAM . . . 299
Main Memory/DRAM . . . 299
Storage . . . 299
Fault-Tolerant On-Chip Communication . . . 300
Cross-Layer Reliability . . . 301
Domain-Specific Fault Tolerance . . . 303
Fault Tolerance in Emerging Technologies . . . 304
Emerging Memory Technologies . . . 304
Reliability Issues in NVMs . . . 306
Fault Tolerance in AI/ML . . . 309

S. S. Sahoo · A. Kumar (✉)
Technische Universität Dresden, Dresden, Germany
e-mail: siva_satyendra.sahoo@tu-dresden.de; akash.kumar@tu-dresden.de

A. Das
Drexel University, Philadelphia, PA, USA
e-mail: anup.das@drexel.edu

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_11

Conclusion 314
References 316

Abstract

Fault-tolerant computing has been the cornerstone of reliable computing using electronic systems. Traditionally, fault-tolerant system design has been primarily driven by the system's operating environment and the resulting fault scenarios due to external disturbances. However, with the growing unreliability of modern semiconductor technologies, reliable computing requires designing fault tolerance across multiple layers of the system stack. Further, the growing impact of intrinsic fault mechanisms such as aging requires designing fault-tolerant architectures across application domains. To this end, emerging technologies, both semiconductor devices and applications, have brought about novel challenges to designing fault-tolerant architectures.

This chapter provides a brief overview of the landscape of fault-tolerant architectures, from the fundamentals to the state-of-the-art and open research areas. The chapter begins with the background of faults, errors, and reliability estimations. Fault-tolerant architecture for computation, memory/storage, and communication are briefly covered. Related state-of-the-art topics such as cross-layer reliability and fault-tolerance for emerging devices (NVMs) and emerging applications (AI/ML) are also covered in the chapter.

Keywords

Fault-tolerant computing · Reliability-aware system design · Computer architecture · Reliability in AI/ML · NVM reliability · Fault-tolerant architectures · Cross-layer system design

Introduction

From deciphering Lorenz-encrypted messages to rendering of virtual environments for the metaverse, computing systems have witnessed exponential growth in both the quantity and variety of computation workloads. Moore's Law (Moore 2006) has been one of the major driving forces enabling electronic systems to cater to such increasing computation demands. As shown in Fig. 1, the last 50 years have witnessed near exponential growth in the number of transistors on an integrated circuits (IC). Such growth has been possible primarily due to the aggressive semiconductor technology scaling. Technology scaling, along with architectural innovations has also led to increasing operating frequency, at least until 2005. However, the reducing transistor sizes along with the increasing power density (owing to higher clock frequency) have also resulted in increasing manufacturing defects and reduced robustness to external faults. Further, continued efforts toward supply voltage scaling, in order to reduce power density, have exacerbated the

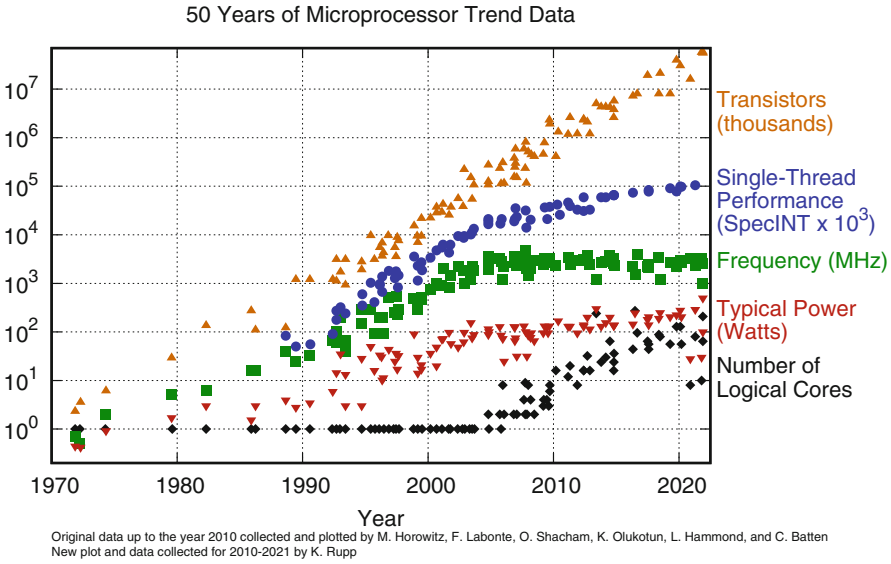


Fig. 1 50 years of processor trends

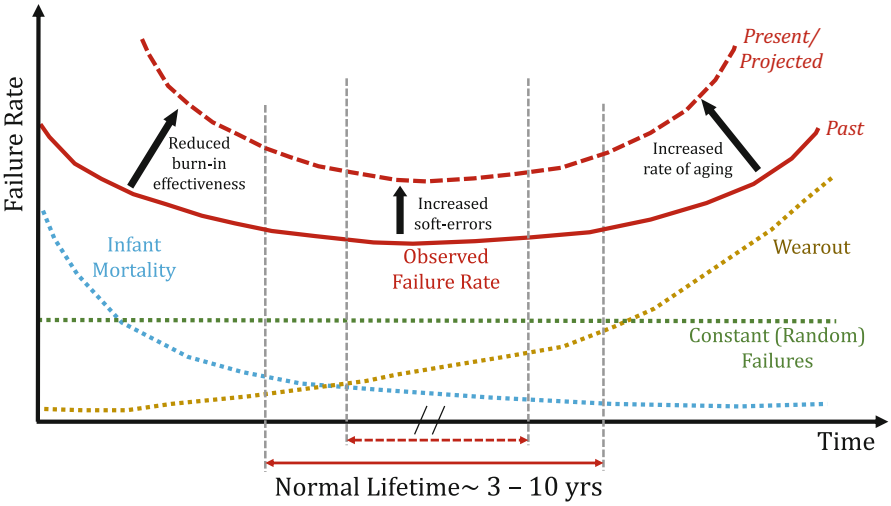


Fig. 2 Increasing unreliability in electronic systems

effects of external disturbances. While the logical masking of such externally induced soft errors has remained unchanged, the electrical masking and latching window masking have reduced due to the supply voltage scaling and higher clock frequencies, respectively.

This increasing unreliability of electronic systems is captured in the *bathtub* curve, shown in Fig. 2, showing the effect of increasing faults in the three stages

of typical electronic systems' life cycle, namely, *infant mortality*, primarily caused by premature failure of weak components as a result of manufacturing defects and burn-in testing and *constant failures* due to random faults and the *wearout-based faults* due to aging. The solid curve in the figure shows the net effect of all three factors. With aggressive technology scaling, the rate of manufacturing defects has increased, resulting in higher infant mortality and higher susceptibility to aging-related faults. As mentioned earlier, the higher soft error rate results in increasing constant failure rate due to random faults. Further, the increasing power density due to higher clock speeds and parallel processing (in multicore systems) results in accelerated aging. The net result of all these factors is the increased failure rate as shown by the dashed bathtub curve in the figure.

The effect of increasing faults can be observed across all types of architecture components – computation, communication and storage – resulting in the degradation of different Quality of Service (QoS) metrics of an application. If unmasked, soft errors in memory elements and interconnects result in reduced *functional reliability*. Similarly, degradation in interconnects and logic elements may lead to slower components leading to reduced *timing reliability*. *Lifetime reliability* of electronic systems is adversely affected by both aging and permanent faults of the components.

Redundancy has been the primary approach for achieving fault tolerance across the computation stack. It usually involves replicating execution across (1) multiple components (*spatial*), (2) a single component (*temporal*), and (3) additional data (*information*) to detect and/or mask faults. The implementation cost and efficacy of each type of redundancy may vary for different applications, and given the system's constraints, a subset of such methods may prove to be infeasible for the application. Further, depending upon the application domain, a system may prioritize one/more among functional, timing, and lifetime reliability. For instance, in real-time systems, the timeliness of execution has the highest priority. Similarly, in financial and scientific computations, the accuracy of calculations would be more important than the execution time. Additionally, in systems such as consumer products and space missions, extended lifetime of the system may have higher priority. Also, in a system executing multiple applications, each application may have varying criticality w.r.t. each reliability-related performance metric. In this scenario, a uniform approach to fault tolerance can result in under-/overdesign. Therefore designing fault-tolerant architectures forms an important aspect of application-specific computing.

This chapter briefly covers the various aspects of fault-tolerant architectures – background, taxonomy, methods, and the state of the art. The rest of the chapter is organized as follows. Section "Faults, Errors, and Failures" provides a brief overview of the fault mechanisms followed by a description of the relevant nomenclature and taxonomy. Section "Fault Tolerance" covers the generic methods and methodologies involved in fault tolerance. More specific approaches to fault-tolerant designs for computation, memory/storage, and communication are presented in sections "Fault-Tolerant Computation", "Fault-Tolerant Memory/Storage", and "Fault-Tolerant On-Chip Communication", respectively. Sections "Fault Tolerance in Emerging Technologies" and "Cross-Layer Reliability" present more recent advancements in fault-tolerant architecture w.r.t. emerging technologies

and cross-layer reliability, respectively. Finally, we conclude the chapter in section "Conclusion" with a discussion on the scope of related research topics.

Faults, Errors, and Failures

Fault Model

The events in a system related to fault tolerance can be classified as one of *failure*, *error*, and *fault* (Avizienis et al. 2004). An application failure refers to an event where the service delivered by the system deviates from the expected service defined by the application requirements. An error refers to the deviation of the system from a correct service state to an erroneous one. Faults refer to the adjudged or hypothesized cause of the error. The faults in any computing system may be caused by either physical faults affecting the hardware or due to imperfect software implementation. For the current chapter, we limit the discussion to physical faults only. A major classification of faults is based on the frequency and persistence of the occurrence of the faults:

- *Transient Faults* occur at a particular time, remain in the system for some period, and then disappear. Such faults are initially dormant but can become active at any time. Examples of such faults occur in hardware components that have an adverse reaction to some external interference, such as electrical fields or radioactivity.
- *Intermittent Faults* show up in systems from time to time due to some inherent design issue or aging. An example is a hardware component that is heat-sensitive – it works for some time, stops working, cools down, and then may start to work normally again.
- *Permanent Faults* such as a broken wire or a software design error show a more persistent behavior than intermittent faults – start at a particular time and remain in the system until they are repaired.

While the above classification is based on the cause of the physical faults, different fault models are used for investigating the effect of such faults at higher abstractions. Such fault models include:

- *Stuck-at fault*: This is used to model the effect when a memory cell or the input/output of a logic gate is permanently fixed to a single logic value – stuck-at-zero or stuck-at-one.
- *Single bit-flip fault*: This is used to model the transition from one logic value to another. Such a model can also be used to represent an instance of a bit flip, for example, as a result of fault injection, or a bit flip within a time window, where the original logic value is restored after the fault duration.
- *Multiple bit-flip fault*: This is used to model the logic transition (due to a fault) across multiple bits. One of the primary use-cases for such models includes representing faults due to coupling – electric shorts and electromagnetic.

Fault Mechanisms

Several fault mechanisms are responsible for causing one or more of the above types of faults. The mechanisms of physical faults can be broadly categorized as discussed below.

External Faults

Investigation into external factors, primarily cosmic radiations, causing anomalous behavior in digital circuits have been reported since 1975 (Binder et al. 1975). Such factors can result in both transient and permanent faults. Transient faults can lead to *soft errors* which usually refer to non-reproducible hardware malfunctions. The additional charge induced by external interference (such as alpha particles from radioactive impurities in chip packaging materials and neutrons generated by cosmic radiation's interaction with the earth's atmosphere) can sometimes lead to changing the logic value. As shown in Fig. 3, the interaction of these particles and the silicon lattice, on striking the IC, produces a *collected charge* which might be sufficient – greater than the critical charge, Q_{crit} – to change the logical value of the impacted node. While in memory elements, the changed value is retained until the next refresh; in combinational circuits the computations are affected only if the wrong value is latched by a memory element.

Permanent faults can be caused by radiation effects such as Single Event Latchup (SEL) and Total Ionizing Dose (TID) that the device is exposed to. SEL refers the phenomenon when the passage of a single particle induces the creation of parasitic bipolar (p-n-p-n) shorting of power to ground. TID refers to the cumulative long term ionizing damage due to protons and electrons and can cause devices to suffer threshold shifts, increased device leakage, power consumption, timing changes, decreased functionality, etc. Although the voltage scaling has reduced the probability of occurrence of SELs, the reduced feature size in advanced technology nodes and the resulting manufacturing defects have increased the vulnerability to TID (Arzt et al. 1992).

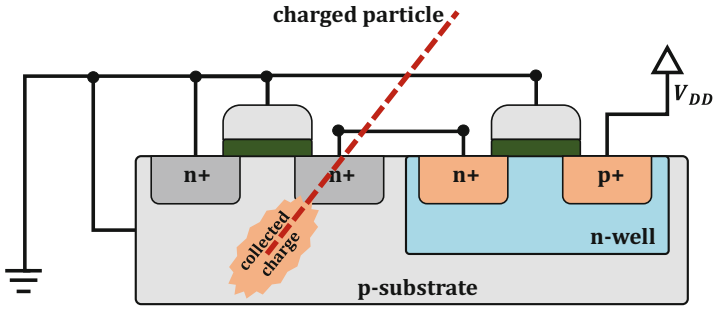


Fig. 3 Charge accumulation by radiation

Aging/Stress-Induced Faults

The term aging broadly refers to the degradation of semiconductor devices due to continued electrical stress that may lead to timing failures and reduced operational life of the IC. The primary physical phenomena causing aging are shown in Fig. 4 and are listed next:

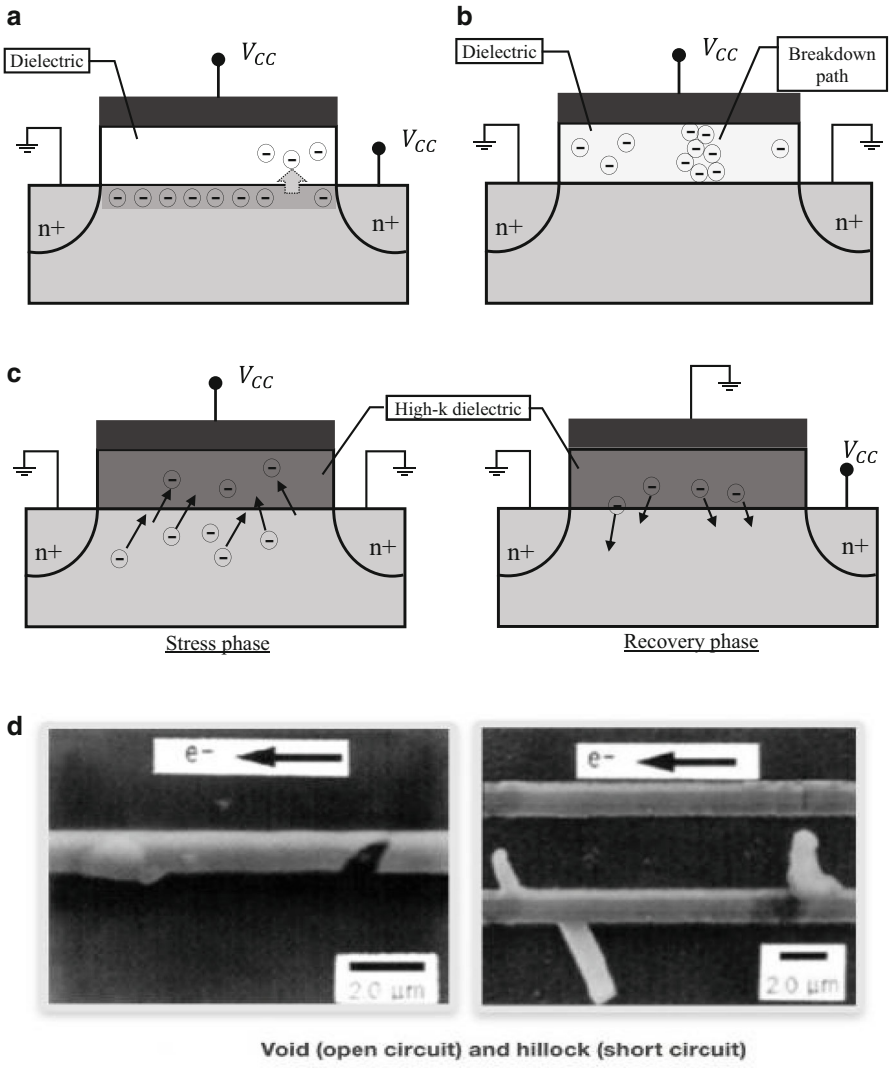


Fig. 4 Aging-related fault mechanisms. (**a**) HCI in NMOS. (**b**) TDDB in NMOS. (**c**) Positive bias temperature instability (PBTI). (**d**) Open and short circuit faults due to electromigration. (Source: W.D. Nix et al. 1992)

1. Bias Temperature Instability (BTI) results in an increase in the threshold voltage, V_{th}, due to the accumulation of charge in the dielectric material of the transistors. The use of high-k dielectrics in lower-technology nodes has resulted in an increased contribution of BTI to aging.
2. Hot Carrier Injection (HCI) occurs when charge carriers with higher energy than the average stray out of the conductive channel between the source and drain and get trapped in the insulating dielectric. Eventually it leads to building up electric charge within the dielectric layer, increasing the voltage needed to turn the transistor on.
3. Time Dependent Dielectric Breakdown (TDDB) comes into play when a voltage applied to the gate creates electrically active defects within the dielectric, known as traps, that can join and form an outright short circuit between the gate and the current channel. Unlike the other aging mechanisms, which cause a gradual decline in performance, the breakdown of the dielectric can lead to the catastrophic failure of the transistor, causing a malfunction in the circuit.
4. Electromigration (EM) occurs when a surge of current knocks metal atoms loose and causes them to drift along with the flow of electrons. The thinning of the metal increases the resistance of the connection, sometimes to the point that it can become an open circuit. Similarly, the accumulation of the drifted material can lead to electrical shorts.

The fault mechanisms described above primarily refer to operating environment/workload-induced faults. However, fault injection methods may also be deliberately used to result in transient and permanent faults (Bar-El et al. 2006). While some of such fault injection methods may aid in testing the vulnerability of the system, most of the related works has focused on inducing faults in semiconductors to exploit the resulting errors maliciously. Such induced faults can be either provisional, similar to transient faults, or destructive, similar to permanent faults. The techniques for deliberate fault injection include variation in the supply voltage or external clock to generate glitches in order to induce misinterpretation and/or skipping of instructions. Similarly, thermal, optical, and electromagnetic field-based fault attacks are used to alter memory bits.The first fault injection-based attack targeted the RSA public cryptosystems. Similar attacks on other cryptosystems have also been reported. Interested readers can refer to Karaklajić et al. (2013) for detailed fault attacks and some related countermeasures. In the current chapter, we will limit the scope of discussion of fault tolerance to environment/workload-induced faults only.

Fault Masking

The fault mechanisms discussed above can result in bit flips in memory and/or in some logic node. However, not all faults may lead to errors or failures. Faults can eventually lead to application failure only if they are not masked at any stage. The

masking of faults (and errors) can be attributed to one or more of the following phenomena.

- *Logical Masking*: The inputs to the combinational circuit or computation do not allow a logical path between the node affected by the Single Event Upset (SEU) and the output. As shown in Fig. 5a, in a half-adder circuit, any incorrect logic value at one of the inputs of the "and" gate does not impact the carry-out value as long as the other input is at logical 0. Such masking effects are usually not affected by technology scaling and are application-/design-specific.
- *Electrical Masking*: The changed pulse due to an SEU is attenuated during propagation and is too small to be captured. Continuous voltage scaling in advanced technology models has led to decreasing Q_{crit} and higher susceptibility to transient faults by external charged particles, leading to higher Soft Error Rate (SER).
- *Latching-window Masking*: The affected pulse arrives too early or too late to be captured by the storage element. As shown in Fig. 5b, deeper pipelines and the resulting higher clock frequencies tend to increase the probability of the incorrect logic value being latched by the storage elements. Hence, there has been a reduction in the latching window masking of transient faults, leading to higher SER.

The fault masking described above refers to more implicit phenomenon that may occur in a system. Additional fault mitigation methods are usually implemented at multiple abstractions for improving the fault tolerance of the system. Given the implicit fault masking and fault mitigation methods, Fig. 6 shows the different scenarios that can occur as a result of a hardware fault. As seen in the figure, an error occurs in the system only when a wrong logic value is either read from a storage bit or latched from a gate output and not mitigated by any error protection mechanisms. Similarly, an application failure occurs only when the error is not masked by the application logic and the resulting deviation is beyond the application's tolerance limits. Estimation of the error and failure probabilities of a system are covered in section "Reliability Estimation."

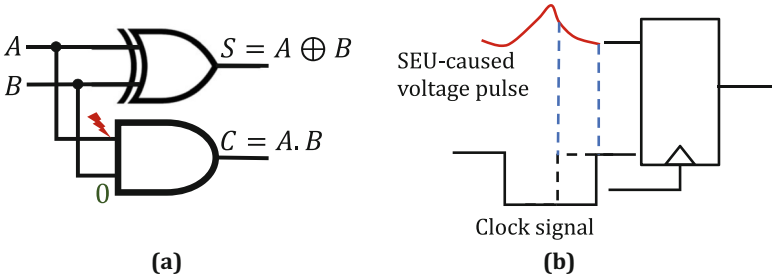


Fig. 5 Fault Masking: (**a**) Logical masking in half-adder combinational circuit. (**b**) Reduced latching-window masking due to faster clocks

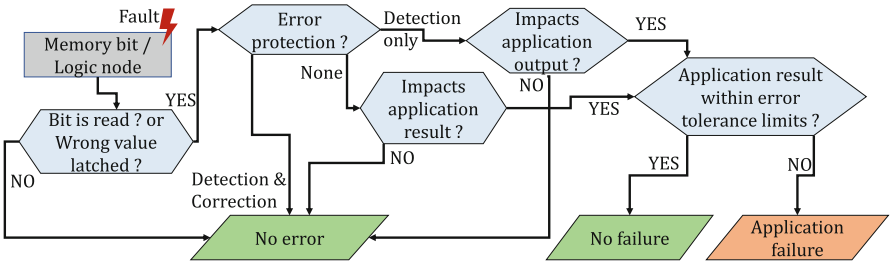


Fig. 6 Fault and error masking

Reliability

Reliability is one of the fundamental attributes of dependable computing and describes the ability of the system to deliver its intended service (Dubrova 2013). Mathematically, reliability $R(t)$ at a time instance t represents the probability of the system to perform without failure in the time range $[0, t]$. Evidently, the reliability of the system is strongly related to the fault tolerance designed into the system and resulting application failure behavior.

Types of Reliability

As shown in Fig. 6, an application's failure rate is correlated to the application-specific tolerance to errors and resulting deviation in application behavior. The type and extent of such tolerances may vary with the application domain and are characterized by application-specific QoS requirements. The fault tolerance specific QoS metrics (some of which are shown in Fig. 7) can be categorized as follows:

- *Functional Reliability*: With the rising constant failure rates during the normal life of the system, the chances of such failures manifesting as incorrect computations have also increased. Hence, in applications that require high levels of computational accuracy such as financial transactions in point-of-sales systems or ATMs, scientific applications, the corresponding QoS can be expressed in terms of functional reliability. It concerns the correctness of the results computed by a system operating in a fault-inducing environment. The functional reliability can be quantified by the probability of no physical fault-induced errors occurring during application execution or the Mean Time between Failures (MTBF).
- *Timing Reliability*: The performance of the system in terms of the expected behavior concerning the timeliness of execution completion can be expressed as its timing reliability. It is used only in terms of real-time systems and depending upon the criticality of the application, can be expressed in terms of Worst-Case Executing Time (WCET), MTBF, Probability of Completion and Average Makespan. WCET is usually used in hard real-time systems such as pacemakers and automobile safety features where any instance of missing a deadline can have fatal consequences. MTBF, frequently used in the context of repairable systems,

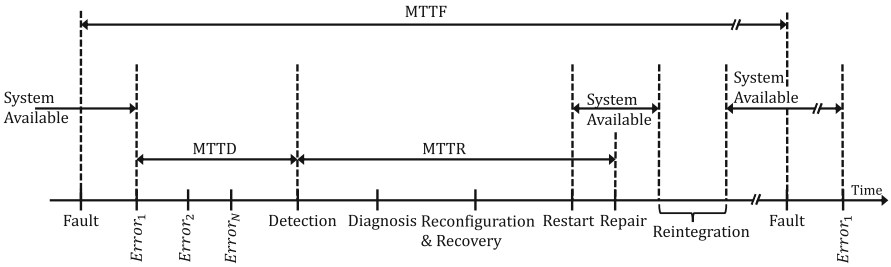


Fig. 7 Reliability metrics

can also be used for expressing the timing reliability in firm real-time systems such as manufacturing assembly lines, where infrequent failures can be tolerated, provided sufficient availability is ensured. Average makespan and probability of completion are usually used in soft real-time systems such as streaming devices and gaming consoles where frequent deadline misses can be tolerated as long as they do not affect user experience.

- *Lifetime Reliability*: The expected operational life of the system can be characterized by its lifetime reliability. Depending upon whether the system is repairable and the cost of such repairs, metrics such as Mean Time To Detection (MTTD), Mean Time To Failure (MTTF), Mean Time To Crash (MTTC), and Availability can be used to characterize the system's lifetime reliability. MTTF refers to the expected time to the first observed failure in the system. In healthcare applications and consumer electronics, the need for predictable and extended MTTF can be the primary objective. Similarly, MTTC refers to the expected operational time for the point at which the system does not have sufficient resources for ensuring the expected behavior and is usually applicable for repairable systems. In applications with long mission times such as space exploration, repairing the failure mechanism is used to extend the MTTC. However, repair time plays a critical role in high-availability applications such as automated control of power generation.

Reliability Estimation

Estimating the implicit fault-masking and the impact of additional fault mitigation measures forms an integral component of designing fault-tolerant system. While underestimation of the impact of faults can lead to unreliable system behavior, overly pessimistic estimations can result in overdesigning, eventually leading to infeasible designs in resource-constrained systems. Some of the more widely used reliability estimations are described below. A more detailed overview of reliability estimation can be found in Wang and Chattopadhyay (2017).

1. **Fault Injection (FI)** involves introducing faults/errors in a system to validate the device dependability under faulty conditions. FI techniques aid in tracking the propagation of faults along with benchmarking systems with high-level FI techniques. Based on their implementation, hardware-related FI may involve:

   - *Physical FI* involves using physical sources such as neutron flux and the pins of the processor to inject faults. The contact-based techniques include using active probes and sockets and Metal-Oxide Semiconductor (MOS) transistors to introduce stuck-at, bridging, open faults, and power line disturbances, respectively. Similarly noncontact methods include using heavy-ion radiations, electromagnetic fields, and focused optical beams to introduce transient faults.
   - *Simulated FI* involves using the simulation infrastructure to inject faults in the design. Since this approach does not require a physical chip, it is much cheaper to implement than physical FI and can be used during early design stages. Simulated FI usually involves changing the run-time state of the simulator through either simulator commands or code modifications (Baraza et al. 2002). Code modification methods, unlike simulator commands, require recompilation and are, hence, costlier. The Simulated FI methods can be implemented at gate/Register Transfer Level (RTL) level, with Hardware Description Language (HDL) models or system-level, with C++ or SystemC models.
   - *Emulated FI* involves introducing faults into an emulation run, typically executing on Field-Programmable Gate Arrays (FPGAs), of the system. Along with providing the fast execution speed of physical FI, it combines the observability and controllability of simulated FI. Various works have leveraged different aspects of FPGAs, such as Dynamic Partial Reconfiguration (DPR), combined HW/SW implementation, SEUs in the configuration memory, etc. to achieve efficient emulated FI.

2. **Analytical Reliability Estimation** usually involves using mathematical models and statistical data for the fast analysis of system behavior under faults. Unlike FI methods, analytical estimation does not require experimental setups or simulation/emulation platforms. Some of the more widely used analytical estimation methods are briefly described below.

   - A*rchitectural* V*ulnerability* F*actor (AVF)* was proposed by Mukherjee et al. (2003) as a method to calculate the probability of a fault occurring in the architecture affecting the user-visible outputs. This calculation is based on tracking the subset of processor state bits that are required for Architecturally Correct Execution (ACE). In the absence of any error correction methods, a fault in a storage cell containing the ACE bits results in an error. The estimation involves determining the *un-ACE* bits, processor state bits not necessary for ACE, at both architecture and micro-architecture levels. The authors report un-ACE bits arising from NOP instructions, performance-enhancing instructions (e.g., prefetches), predicated-false instructions, dynamically dead

code, and logical masking at the architecture-level. Similarly, at the micro-architecture level, the authors report un-ACE bits from idle or invalid bits; mis-speculated bits, such as wrong-path instructions; predictor structure bits; and microarchitecturally dead bits. The AVF analysis has been successfully used in avoiding pessimistic error estimates resulting in over designing. Further the AVF approach has been used to determine the vulnerability factor at higher abstractions, namely, Instruction Vulnerability Index (IVI) and Function Vulnerability Index (FVI) to explore design trade-offs between timing and functional reliability (Rehman et al. 2016).

- *Probabilistic Transfer Matrix (PTM)*, proposed by Krishnaswamy et al. (2008), provides a methodology for representing probabilistic behavior in logic circuits. Primarily a circuit-level technique, PTMs provide an algebraic representation of logic circuits and can be used to model a wide variety of faults – both deterministic and probabilistic. Further PTMs can be used to model glitch propagation and provide an efficient method to formulate functions such as circuit fidelity. Krishnaswamy et al. (2008) represent the truth-table of a gate-level circuit as a matrix containing only 0s and 1s, the ideal transfer matrix. With the PTM, the columns of the matrix, representing the inputs, are allowed to have real values in the range [0, 1], thus representing the error probabilities of each input. PTMs, although very accurate, do not scale well for more complex circuits as the matrix sizes start becoming infeasibly large.
- *Timing and Lifetime Reliability*: Analytical estimation of timing reliability usually involves determining the probability of the execution completing before a deadline. While the estimation with deterministic execution can be fairly simple, the complexity arises due to probabilistic execution times resulting from resource contention or replicated execution. Works such as Sahoo et al. (2018b); Sahoo et al. (2020b,a) have presented various methods of analytical estimation of timing reliability under different probabilistic execution scenarios. The lifetime reliability estimation usually follows the principle that in the absence of any component level redundancy, a structure's reliability depends on the first component's failure. A detailed analytical estimation approach to device, component, and architecture-level lifetime reliability, across different fault mechanisms, is presented by Xiang et al. (2010).

Fault Tolerance

In order to achieve the reliability-oriented QoS requirements of a system, appropriate fault tolerance measures need to be implemented. Such measures can include fault/error avoidance, detection and mitigation for transient faults, diagnosis and repair for permanent faults, etc. These measures can be implemented at one/multiple layers of the computation stack. Redundancy forms a primary tool for implementing fault tolerance measures. Redundancy usually involves complementing the regular

computation with additional processing in order to detect and/or mask the effect of errors during computation. Next we take a brief look at some generic fault tolerance activities and the types of redundancies used to implement them.

Fault Tolerance Activities

The different activities involved in fault tolerance are shown in Fig. 8. Depending upon the repairability of the system, implementing fault tolerance may involve a subset of the following:

- *Fault avoidance*: This usually involves building the system with robust components that are less susceptible to faults. It may also include avoiding system configurations that increase the susceptibility to fault mechanisms. For instance, given the impact of Dynamic Voltage Scaling (DVS) on SER, the designer may choose to avoid configurations with lower supply voltages in high-radiation environments.
- *Detection*: This is primarily to ascertain whether the computation has been affected by any kind of faults. Depending upon the implementation, it may incur timing and/or resource overheads.
- *Diagnosis*: Unlike detection, diagnosis concerns determining the fault characteristics. This includes finding the type of fault – transient or permanent, locating the component(s) causing the fault, estimating the impact of the fault on application behavior, etc.
- *Isolation*: This process forms an integral component of repairable systems and includes eliminating the faulty components from the computation until repair is completed. In case of permanent faults, this may include finding the components (or sub-components) that need to be excluded from any future computation.
- *Repair*: In repairable systems, fault tolerance may include allowing the affected components to recover partially/completely to their operational states. For instance, it may include reducing the workload on faster aging components to reduce thermal stress.

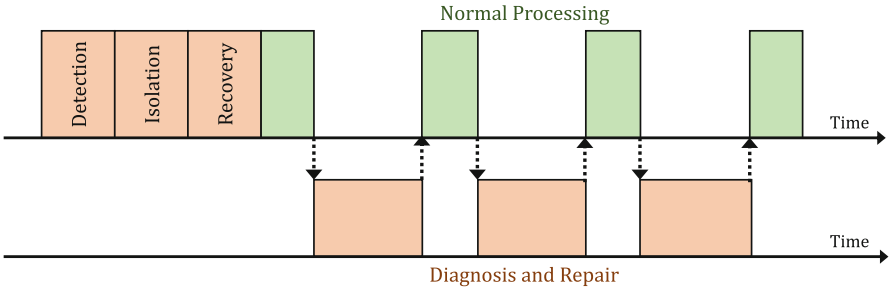


Fig. 8 Fault and error masking

- *Recovery*: While repair usually concerns the components of the system, recovery primarily concerns the application execution. It can range from the mitigation of any detected faults/errors to resetting the system from a *hang* state.

Redundancy

Redundancy can be defined as *the replication of critical components and/or functions in a system with the primary goal of improving the system's reliability*. For fault tolerance, different layers of the system may implement various forms of replication. Broadly, such measures can be categorized under three types of redundancies – spatial/physical, temporal, and information. Depending upon the extent to which each of these is implemented, it can result in varying effectiveness and overheads. We discuss each of the redundancy types in the context of a matrix multiplication and compare their impact on the functional, timing and lifetime reliability. Figure 9 shows the 4×4 matrix multiplication with the Multiply and Accumulate (MAC) considered as the computation module. Assuming the multiplication of two $N \times N$ matrices, the MAC would compute N^2 dot products. We consider the computation of each element of the output matrix as a single operation. Also, for the error model, we assume an average fault rate of λ following a Poisson distribution. Correspondingly, the probability of an error occurring with increasing execution time is depicted in Fig. 10.

- *Spatial/Physical Redundancy* usually involves using additional hardware modules for replicating the computation. As shown in Fig. 11a, such replication could be targeted for detection of errors by using Dual Modular Redundancy (DMR). Similarly, Triple Modular Redundancy (TMR) can be used to mask the effect of any errors in one of the modules, as shown in Fig. 11b. The voting module usually implements a majority function and can add an insignificant amount of timing overhead, as shown in Fig. 14. Figure 11c shows a more generalized version – *N*-Modular Redundancy (NMR), where the result is considered acceptable iff at-least *k*-modules provide the same output: a *k-out-of-N:G* system. For TMR (a 2-out-of-3:G system), the probability of error can be estimated as $3 \times p^2_{error}(t) \times p_{ne}(t) + p^3_{error}(t)$, which corresponds to the scenario where all three or any two of the three modules encounter an error during computation. It can be noted that the NMR methods shown in Fig. 11 aim to improve the

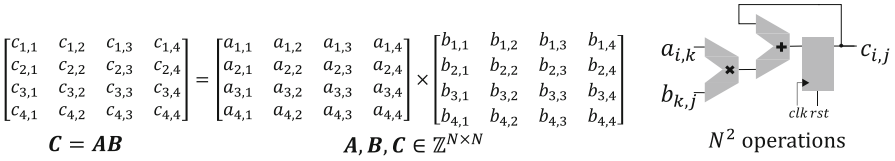


Fig. 9 Matrix-matrix multiplication

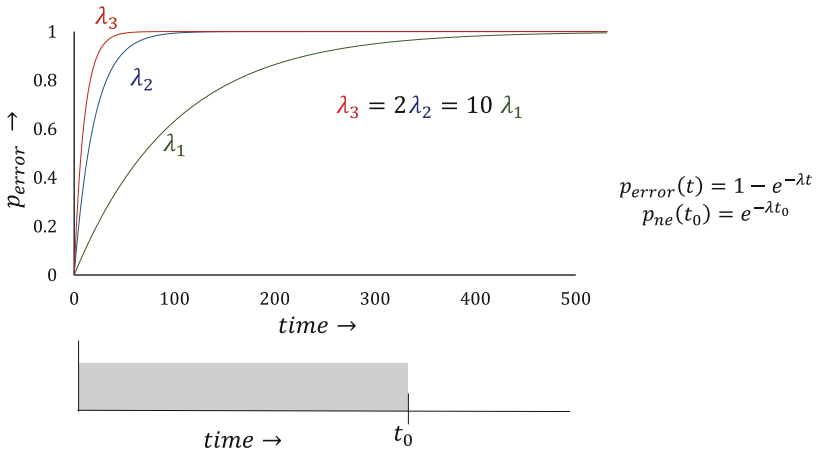


Fig. 10 Error probability

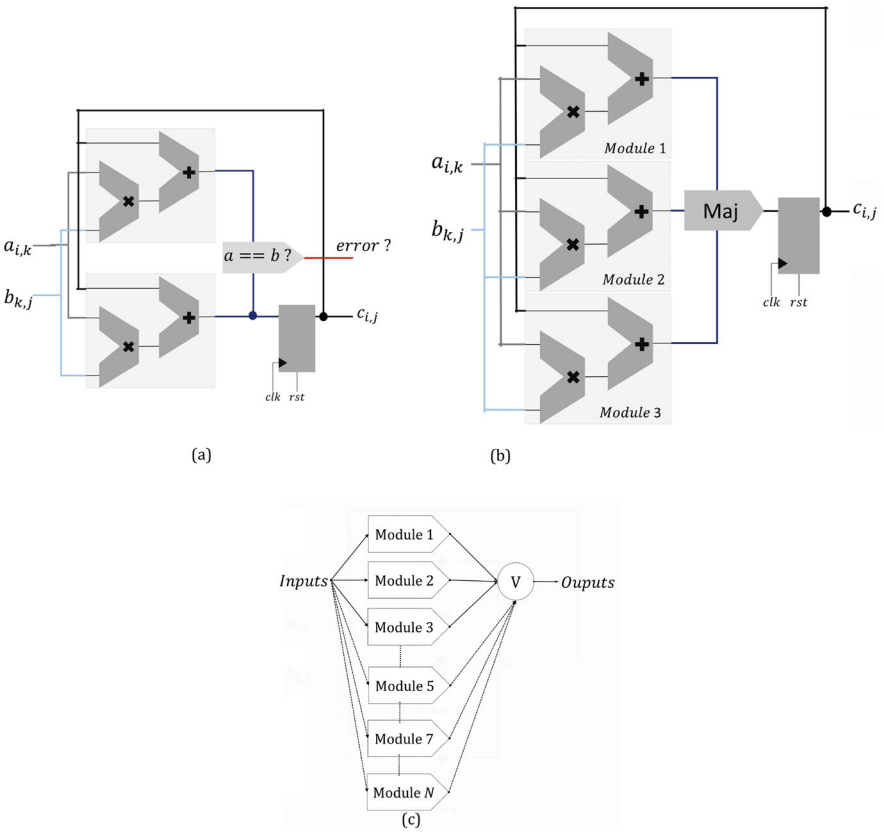


Fig. 11 Spatial redundancy

functional reliability but may adversely affect the lifetime reliability owing to the parallel execution and resulting higher thermal stress. Spatial redundancy for improving the lifetime reliability usually involves using cold spares.

- *Temporal Redundancy* involves additional re-execution of the computation workload on the same hardware module. As shown in Fig. 12a, it may include complete re-execution when an error is detected at the end of the computation. Similarly, with parallel error detection, the re-execution may occur from different points, depending on the error occurrence, as shown in Fig. 12b. Figure 12c shows the more widely used method of check-pointing with roll back recovery. In the context of the matrix multiplication, it may involve creating checkpoints after the computation of each row of the output matrix, such that any re-execution can begin by restoring the stored checkpoint. The functional reliability of such methods is usually determined by the accuracy of the detection methods. As shown in Fig. 13, the timing reliability of temporal redundancy-based methods requires more complex estimation approaches as the execution time is nondeterministic in such cases. Since temporal redundancy usually results in increased computation per unit workload, it eventually leads to a lower operational lifetime.
- *Information Redundancy* involves computation of additional data points in order to detect and/or recover any errors in computation. For instance, as shown in Fig. 13, the input matrices are augmented with their column and row checksums resulting in larger operands A_{cc} and B_{rc}, respectively. Correspondingly, the output matrix is a full checksum version of the original output matrix: C_{fc}. As a result, it requires $(N+1)^2$ operations instead of N^2 and, therefore, adversely affects both the timing and lifetime reliability.

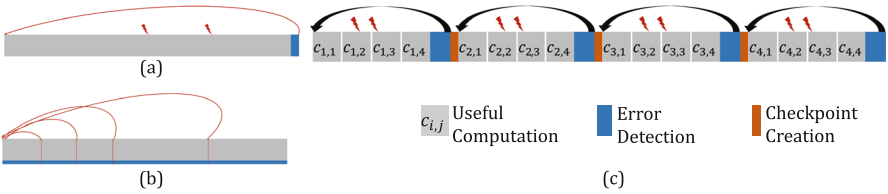


Fig. 12 Temporal redundancy

$$\begin{bmatrix} c_{1,1} & c_{1,2} & c_{1,3} & c_{1,4} & c_{1,5} \\ c_{2,1} & c_{2,2} & c_{2,3} & c_{2,4} & c_{2,5} \\ c_{3,1} & c_{3,2} & c_{3,3} & c_{3,4} & c_{3,5} \\ c_{4,1} & c_{4,2} & c_{4,3} & c_{4,4} & c_{4,5} \\ c_{5,1} & c_{5,2} & c_{5,3} & c_{5,4} & c_{5,5} \end{bmatrix} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \\ a_{5,1} & a_{5,2} & a_{5,3} & a_{5,4} \end{bmatrix} \times \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} & b_{1,5} \\ b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} & b_{2,5} \\ b_{3,1} & b_{3,2} & b_{3,3} & b_{3,4} & b_{3,5} \\ b_{4,1} & b_{4,2} & b_{4,3} & b_{4,4} & b_{4,5} \end{bmatrix}$$

$$C_{fc} = A_{cc} \times B_{rc}$$

C_{fc}: $(N+1)^2$ operations,
+ Error detection,
+ Error correction

$$a_{N+1,j} = \sum_{i=1}^{N} a_{i,j} \quad for\ 1 \le j \le N$$

$$b_{j,N+1} = \sum_{j=1}^{N} b_{i,j}, \quad for\ 1 \le i \le N$$

Fig. 13 Information redundancy

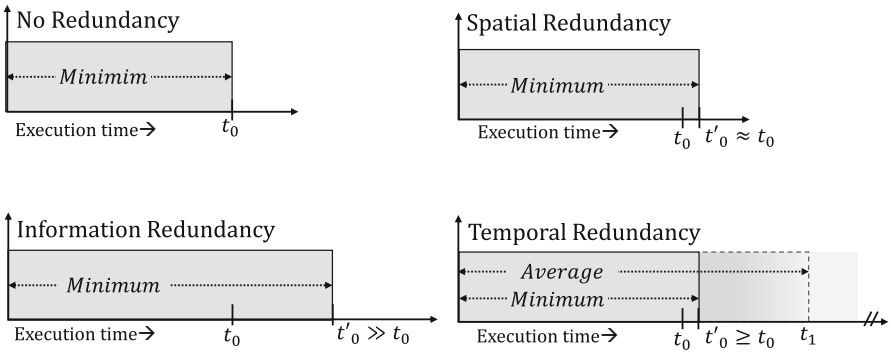


Fig. 14 Redundancy and timing reliability

While redundancy forms a generic approach to providing fault tolerance, depending upon the component type, specialized methods are employed at one or more layers. Further, as shown in Fig. 14, depending upon the type of redundancy, different levels of timing overheads are introduced into the execution. While spatial and information redundancy results in more deterministic timing overheads, temporal redundancy methods may lead to nondeterministic overheads, thereby requiring complex estimation methods. The following sections provide a brief survey of architecture-level fault tolerance methods for computation, memory, and communication.

Fault-Tolerant Computation

The fault mechanisms described in section "Faults, Errors, and Failures" can result in errors in computation and reduced system lifetime. We describe the methods for improving fault tolerance in computation under three categories. Single-core computing includes all methods that can improve reliability of a single Processing Element (PE). The methods that take advantage of the presence of multiple PEs on the architecture are described under multicore computing. The reconfigurable computing subsection describes specialized methods applicable to reconfigurable architectures – FPGA and Coarse-Grained Reconfigurable Arrays (CGRA).

Single-Core Computing

Tolerance techniques for soft errors in computation circuits primarily involve some form of execution redundancy – either spatial or temporal. Implementing DMR provides only fault/error detection, and TMR provides masking of any single fault/error. In terms of cost, TMR can result in more than 200% area and power

overheads. Area and power overheads in a LEON3 core when introduced with varying levels of TMR in pipeline, cache, and register file are reported in Kriebel et al. (2014). Corresponding results for an FPGA-based implementation of LEON3 are presented in Wirthlin et al. (2016). In both cases, power and area overheads of more than 200% are observed. Circuit-level fault masking usually involves circuit hardening by using multiple flip-flops or by gate resizing. Multiple flip-flop-based designs use scan-flops already present in the circuit to provide error tolerance. Gate resizing involves using bigger transistors to provide better tolerance against radiation-induced soft errors. More flexible methods, presented in Mohanram and Touba (2003) and Rao et al. (2006), use partial replication based on profiling results to obtain reduced coverage at lower power and area overheads. Similarly, low overhead methods based on circuit monitoring enable low cost and configurable fault detection.

At the architecture level, the granularity of execution replication provides the trade-off in error resilience and associated overheads. The granularity may vary from a single module like the pipeline (Austin 1999; Slegel et al. 1999) to an entire core in chip multiprocessors (Mukherjee et al. 2002). Time redundancy-based techniques like redundant multi-threading are also used. Some fault detection methods involve manipulating the pipeline (Blaauw et al. 2008) to detect both transient and intermittent faults. Similar to circuit level, symptom or assertion monitoring-based detection methods provide incomplete coverage at very low overheads. The symptoms include exceptions, control flow mis-speculations and cache or translation look-aside buffer misses.

In addition to the uniform protection schemes discussed above, opportunistic protection schemes aim at using underutilized resources to insert redundant execution for critical portions of the computation. Wang et al. (2013) present two protection schemes: an aggressive scheme that achieves redundant execution for all protected instructions while incurring high overheads and an opportunistic protection scheme that replicates protected instructions only if there are NOP instructions around it, thereby incurring zero penalty:

$$\begin{aligned} &A \times N_1 \pm A \times N_2 = A \times (N_1 \pm N_2) \\ &(N_1 \otimes N_2)\ mod\ A = ((N_1\ mod\ A) \otimes (N_2\ mod\ A))\ mod\ A \end{aligned} \tag{1}$$

Some code-based methods are also used at architecture level to provide concurrent fault detection and masking at lower area overheads. These methods are based on *AN codes* (Rao 1974) and are based on the principle of providing a redundant representation of numbers such that the results of some operation on them can be analyzed to detect and correct errors. As shown in Equation(1), some operations preserve certain properties of the operands. Such operations are performed on both operands as well as the result, and the results can be used for detection and sometimes correction. However, such methods are very application-specific and require high design effort.

The redundancy-based methods discussed aim for symmetric reliability, where every component is assumed to have equivalent priority. However asymmetric reliability approaches aim at reducing the overheads by prioritizing more significant portions of the system. For instance, Wang et al. (2014) propose a Divide and Conquer Hamming (DCH) scheme for messages that divides the message into multiple blocks while enabling different protection levels for each block. Similarly, Wang et al. (2016) propose an error confinement method that uses statistical data and Monte Carlo simulations to correct errors in the output by replacing erroneous data with a best-effort estimate rather than using error-correcting circuitry.

Multicore Computing

Multicore systems have built in structural redundancy that can be leveraged to improve fault tolerance. Since the redundant code execution can occur in parallel, the overheads on the execution time can be minimal. Depending upon the level of implementation and the sphere of replication, fault tolerance methods for multicore systems can be discussed under the following categories:

- Core level: This involves parallel execution on two separate cores and comparison of the results to detect/mask any faults/errors. The coupling between the cores can range from tight *lockstepping* (Shubu 2008), which requires synchronized execution on similar cores, to loosely synchronized execution across heterogeneous cores. Such methods require additional hardware structures to ensure the synchronizations and enforce hardware-level determinism.
- Simultaneous and Redundant Threading (SRT): SRT-based methods leverage the Simultaneous Multi-threading (SMT) of the cores to implement asynchronous execution of redundant threads and remove the resource conflict of more tight lockstepping methods. Architectural features such as load-value-queue, store buffer, slack-fetch of caches, and branch outcome queues are used to improve the performance of such methods (Sorin 2009).
- Redundant multi-threading: Software-based Redundant Multi-threading (SRMT) approaches involve compiler-based methods that use leading and trailing threads, created by the compiler, to provide redundant execution. It usually involves compiler transformations such as function and input replication. Although primarily a software-based method, architectural enhancements such as hardware message queues can be used to improve the performance.
- Process-level: Such techniques involve replication of the whole process. Although a purely software-based multicore systems enable parallel execution of the processes and reduce the timing overheads.

Architectural repair methods usually involve some form of reconfiguration of the cores to isolate faulty components. Such reconfiguration may involve rearrangement of the pipeline stages, micro-architectural core salvaging and architectural core salvaging. Similarly, error recovery methods usually involve either Forward Error

Recovery (FErR), reconstructing of execution state from an error-free redundant state or Backward Error Recovery (BErR), or rolling back to saved checkpoints for temporal re-execution. While FErR can leverage parallel redundant execution (e.g., TMR) across multiple cores, BErR can benefit from parallel execution of saving the checkpoint and useful execution.

Reconfigurable Computing

Improving functional reliability in FPGAs has mostly been focused on improving the reliability of the configuration bits. Traditional methods such as Error Checking and Correcting (ECC), scrubbing (Santos et al. 2017), and hardware checkpointing (Koch et al. 2007) have been used to provide protection from transient faults in the configuration memory. Additionally, circuit design methods that involve TMR have also been employed for enabling the usage of FPGAs in high-radiation environments. For lifetime reliability improvement, Srinivasan et al. (2008) proposed various phenomenon-specific methods, tailored for each failure mechanism, to mitigate aging in FPGAs. Similarly, multiple generic electrical stress hot-spot reduction techniques for FPGAs have been proposed. This includes a stress-aware run-time wear-leveling approach that leverages DPR in FPGA-based systems; module diversification, to generate multiple accelerator designs with spatially varying aging effects; and using the modules to leverage DPR by periodically swapping accelerators that use different CLBs of the FPGA fabric. Other approaches include combining module diversification and dynamic adaptation to varying aging effects at run-time and a reliability-aware floorplanning methodology along with delay-based aging estimation and run-time reconfiguration.

A joint mitigation methodology using DPR, aimed at both soft errors and permanent faults in FPGAs, was proposed by Dumitriu et al. (2016). As shown in Fig. 15, the proposed architecture uses spare slots (Partially Reconfigurable Region (PRR)) to enable reconfiguration in the event of any fault. The Multimodal Adaptive Collaborative Reconfigurable self-Organized System (MACROS) is designed to minimize single points of failure by distributing functionality and control among multiple components. The Collaborative Macro-Function Units (CMFUs) are designed to be self-contained and collaborate with the distributed communication and control infrastructure to implement applications semiautonomously. While relocation of the CMFUs to spare slots serves as the fault mitigation method, Built-in Self Recovery (BISR) and Built-in Self Test (BIST) are used for recovery and diagnosis. The authors' use of reconfiguration, as a solution to both types of faults, can be costly for timing reliability.

Almost all the research works employing DPR assume using homogeneous PRR (comprising of equivalent amount of FPGA resources). However, as shown in Fig. 16, Sahoo et al. 2018a; 2019 proposed a hardware/hardware partitioning methodology that allows using application-specific heterogeneous PRRs that provided the scope for improving both the latency (average makespan) and reliability (MTTF) in DPR-based systems.

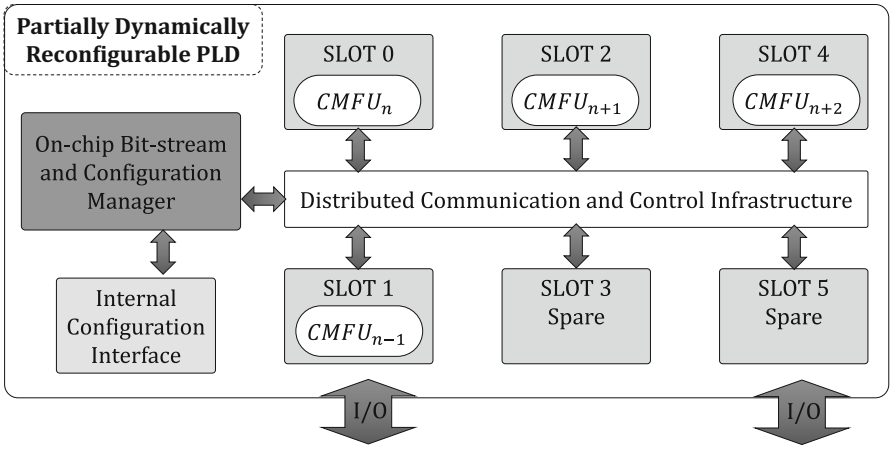


Fig. 15 MACROS architecture (Dumitriu et al. 2016)

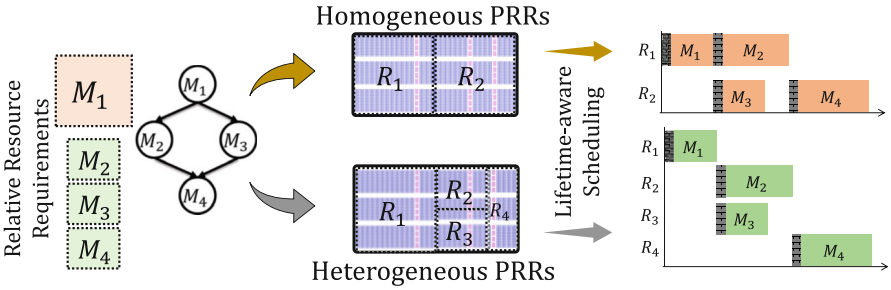


Fig. 16 Reliability-aware HW/HW partitioning (Sahoo et al. 2018a)

Fault-Tolerant Memory/Storage

Information redundancy in the form of additional bits for ECC is commonly used for both Static Random Access Memory (SRAM)-based caches and Dynamic Random Access Memory (DRAM)-based main memory. Hamming or Hsiao code-based Single-bit-Error-Correcting (SEC) and Double-bit-Error-Detecting (DED) codes are usually sufficient for most systems (Hamming 1950; Hsiao 1970). More robust methods such as Double-bit-Error-Correcting (DEC) and Triple-bit-Error-Detecting (TED) codes can be used for higher resilience against random bit errors. Reed and Solomon (1960) codes and Single-Nibble-error-Correcting (SNC) and Double-Nibble-error-Detecting (DND) codes are usually used for protection against multiple-bit burst errors. Granularity of ECC implementation provides the trade-off between resilience and storage overhead. Table 1 shows the storage overheads associated with some ECC implementations.

Table 1 ECC storage overheads (Slayman 2005)

Data Bits	SEC-DED		SNC-DND		DEC-TED	
	Check bits	Overhead	Check bits	Overhead	Check bits	Overhead
16	6	38%	12	75%	11	69%
32	7	22%	12	38%	13	41%
64	8	13%	14	22%	15	23%
128	9	7%	16	13%	17	13%

Cache/On-chip SRAM

In caches, the write-policy determines the amount of correction capabilities that can be implemented. For a write-through policy, the access granularity at Last Level Cache (LLC) from Level 1 (L1) cache is a word, and hence the ECC granularity at LLC should be a single word. However, for a write-back policy of L1 cache, the LLC access is a full L1 cache line and, therefore, allows higher granularity of ECC with reduced overheads. Additional tolerance can be provided by interleaving more ECC codes within the cache line, albeit with more overheads.

Main Memory/DRAM

Most systems use commodity DRAM devices for main memory in Dual in-line-Memory Module (DIMM) or Small Outline DIMM (SODIMM) form factors. ECC DIMMs provide SEC-DED for each DRAM rank and have higher overheads compared to non-ECC DIMMs. More recent methods have been developed to provide fault tolerance against permanent faults in one or more chips on a DIMM. Such Chipkill-correct techniques spread the DRAM access over multiple chips and use single-symbol error-correcting and double-symbol error-detecting codes for error correction. Adaptive methods of ECC like Virtualized ECC (Yoon and Erez 2010) and Bamboo codes (Kim et al. 2015) provide flexible and tunable approaches to main memory fault tolerance. Such techniques can be used to find appropriate trade-offs in cross-layer design approaches. Virtualized ECC uses a scheme where the redundancy information for error detection and correction are stored separately and the correcting information is accessed only in case of a detected error. This reduces the overall power consumption and enables effective error correction and Chipkill-correction for both ECC and non-ECC DIMMs with low overheads.

Storage

With the emergence of data-centric applications such as data analytics and AI/ML, highly reliable storage systems have become indispensable. Unlike the main memory and cache/on-chip SRAM, the reliability requirements in storage systems

are expressed from the perspective of data durability and availability. The faults found in local storage mediums can be categorized into three types. *Whole disk failures* corresponds to situations when a complete storage unit becomes unusable. Latent Section Errors (LSEs) corresponds to the unreachability of select sectors of the storage medium for read/write requests of the application. Undetected Disk Errors (UDEs) can not be repaired by the disk and are only detectable when a read is issued for the affected sector. Correspondingly, the metrics Mean Time to Data Loss (MTTDL) and availability are usually used for quantifying the fault tolerance of storage systems.

Depending upon the type of device used for storage, Hard Disk Drive (HDD) or Solid-state Drive (SSD), specialized fault tolerance measures may be implemented (Kim et al. 2019). However, redundant storage is primarily used for improving fault tolerance, and the implementation can be one of the following methods:

- N-fold data replication: The data is replicated across N-storage targets, and data loss occurs only if the data is corrupted on all storage targets. Evidently, in spite of being highly reliable, this method incurs very high costs.
- K-out-of-N erasure coding: This usually involves implementing some form of information redundancy to transform data of K symbols into N ($> K$) symbols. The transformation ensures recovery of the data from a subset of the N symbols.
- Redundant Array of Inexpensive Disks (RAID): First proposed by Patterson et al. (1988), this involves using multiple physical disk drive components into one or more logical units for improving reliability and/or performance. Depending on the RAID level, one of several schemes may be implemented to satisfy varying QoS requirements: fault tolerance, availability, performance, and capacity. The reliability estimation models for the schemes can be found in Chen et al. (1994).

Storage overhead and power are the cost factors associated with design of a resilient memory system. Flexible error protection methods can enable adaptation to system-level requirements, both at design time and run-time. Summarizing, ECC granularity and fault-coverage provide the tunable parameters and memory controller, the tuning knob for varying error protection levels based on system requirements.

Fault-Tolerant On-Chip Communication

Reliability of the on-chip interconnects in multicore systems involves ensuring proper inter-processor communication by using a combination of different types of redundancies across multiple layers of the Open Systems Interconnection (OSI) model. A detailed account of various reliability issues and the fault mitigation methods can be found in Postman and Chiang (2012). A brief overview of such reliability methods is provided here, limited to Network-on-Chip (NoC)-based communication.

Error detection methods in NoCs include delay sampling for transient faults, and BIST and inline testing for permanent faults. Similarly, the ACK and NACK signals are used in the transport layer to detect errors in transmission. Temporal redundancy-based transient fault mitigation methods for communication include multi-sampling and hop-to-hop retransmission. Similarly, techniques for permanent and intermittent fault mitigation include split-link transmission schemes, where a partially faulty link is used for transmission along with other partially or fully functional links. Similarly, a methodology for improving the lifetime of the communication links is proposed by Das et al. (2013). The proposed method involves joint optimization of the system lifetime of NoC and the processors. Information redundancy-based fault mitigation methods are similar to those used for fault-tolerant memory. However, extra protection is used for the more critical header information of a packet. Forward error correction with block codes and convolutional codes are used for improving the reliability of end-to-end communication. Similarly, probabilistic routing techniques like *flooding*, *gossip flooding*, and *directed flooding* use packet replication and routing across multiple paths to increase the probability of correct transmission. Further, spatial redundancy for communication usually involved using additional and spare wires for transmission (Kakoee et al. 2011).

In addition to the computation, communication, and storage-related fault tolerance methods described above, multiple works have explored the fault tolerance in *uncore* components. Specifically, the Level 2 Cache controllers, the DRAM controllers, crossbar interconnects, and PCI Express I/O controller are studied in Cho et al. (2017) for their impact on application failures due to faults. In addition, Cho et al. (2017) propose replay recovery techniques to reduce the effect of soft errors in uncore components by more than 50× with minimal impact of chip-level area and power overheads.

Cross-Layer Reliability

As discussed in section "Redundancy", redundancy is the primary method for improving fault tolerance across different layers. Unlike the traditional phenomenon-based approach of implementing hardware layer redundancy for each fault mechanism, Cross-layer Reliability (CLR) involves intelligent usage of fault tolerance measures across the system stack (Henkel et al. 2014; Sahoo et al. 2016; Cheng et al. 2016). While the phenomenon-based approach does provide a fault-free abstraction to the other non-hardware layers, the overheads associated with hardware layer redundancy, given the rising fault rates, can be infeasible for resource-constrained systems. In contrast, the cross-layer approach involves multiple layers sharing the fault mitigation activities during run-time (Carter et al. 2010). Similarly, various methods of leveraging cross-layer reliability at both design time and run-time have been proposed (Sahoo 2019).

One of the major advantages of the cross-layer approach is the inherent suitability for application-specific optimizations. Since the overheads of fault tolerance varies with the type of redundancy being used, application-specific tolerances to

degradation in some form of reliability can be used to improve other reliability metrics. Further, with the cross-layer approach, the implicit masking of multiple layers can be used to provide low-cost fault tolerance. Decreasing masking effects in logic circuits was discussed earlier. However, logical masking does not depend on the transistor size and on-chip variations. Therefore, every logic circuit has some masking effect for SEUs. This masking is not limited to logic circuits only. Program-level error masking and its propagation are discussed in Shafique et al. (2013), where the authors exploit program-level masking to perform reliability-driven prioritization of instructions. Experimental studies into the masking effects of software stack (Santini et al. 2015) show that lower application failure rates are observed if more abstraction layers exist between hardware and application. So, the error rates seen by each layer decrease as we abstract away from the hardware layer. However, this benefit is obtained at higher performance overheads. Further, the joint optimization across multiple layers increases the design space considerably.

Recently, there have been multiple works that try to provide efficient Design Space Exploration (DSE) for cross-layer reliability for various system-level design tasks such as task mapping, hardware-hardware partitioning, run-time adaptation, hardware design, etc. (Sahoo 2019; Cheng et al. 2016). However, most of the works assume rather simplistic reliability models such as the one shown in Figure 17 where each layer is limited to a specific type of redundancy. A more holistic approach is designing more realistic reliability models that integrate multiple fault tolerance methods at each layer. As shown in Fig. 18, having reliability interfaces similar to those used for functionality and performance can enable better DSE than current state-of-the-art works. An interface for functional reliability was proposed by Rehman et al. (2016) that used AVF, IVI, and FVI for characterizing different implementations of an embedded processor, instruction set, and function libraries, respectively. However, similar interfaces for timing and lifetime reliability need to be developed for designing efficient cross-layer reliability.

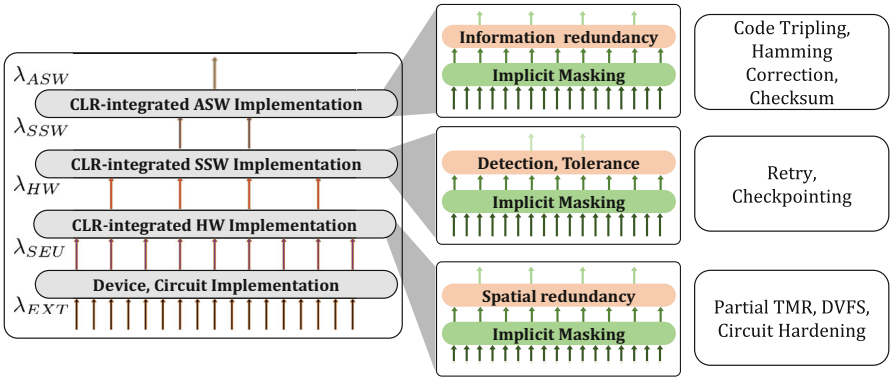


Fig. 17 Redundancy methods across layers. Cross-layer Reliability (CLR) implementation across hardware (HW), System Software (SSW), and Application Software (ASW) (Sahoo 2019)

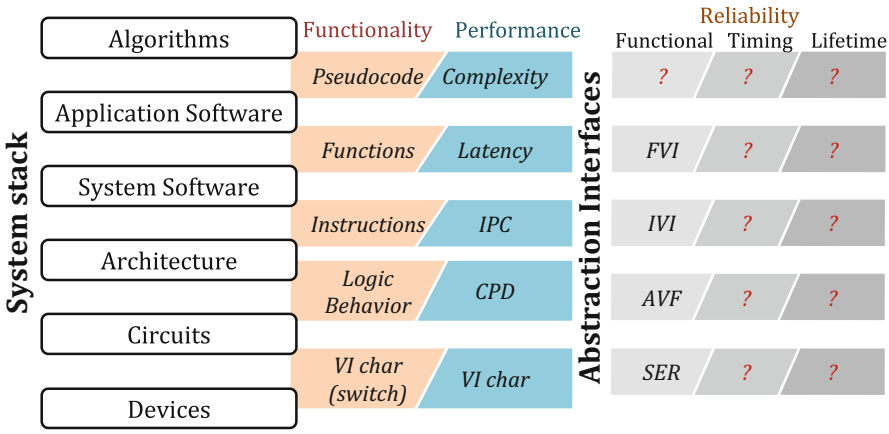


Fig. 18 Interfaces for cross-layer design approach

Domain-Specific Fault Tolerance

With a traditional approach to fault tolerance, radiation-hardened devices can be used for all applications. However, such hardware platforms are far costlier than commercial off-the-shelf devices and are primarily intended for harsh operating environments like high altitudes and outer space. With the rising fault rates, research efforts have targeted using CLR to leverage the application domain-specific characteristics to provide high level of fault tolerance in commercial-off-the-shelf (COTS) devices as well. Some of such multi-layer design techniques related to signal processing and wireless communication are described next.

Signal Processing

Signal processing applications usually involve the acquisition and computations on real world information which has some level of built-in noise. Further, the output from such applications, image processing in particular, are usually used for human perception, which has its own limitations. Therefore, multiple works have leveraged computing paradigms such as approximate computing and precision scaling to provide low overhead fault tolerance. Shim et al. (2004) presented a method of using a reduced precision replica, whose output could be used as the corrected output in case of errors in the original system. Shim and Shanbhag (2006) introduced soft error-tolerant Digital Signal Processing (DSP) by using low complexity estimators of the main DSP block, thereby reducing the overheads of redundancy methods considerably. Biasielli et al. (2022) use the concept of *usefulness* instead of *correctness* of output images to design efficient Convolutional Neural Networks (CNN)-based fault detection. The authors use the CNN-based fault detection along with approximate computing based replicated execution to provide low overhead fault tolerance in a camera image acquisition system. Similarly, Schmidt and French (2013) proposed a combination of HW/SW fault tolerance by using *Radiation*

Hardening by Software and *FPGA Fabric Checkpoint/Restart* to provide fault-tolerant *Fast Lossless prediction* (during image compression) on COTS FPGAs. Other inherently fault-tolerant computing paradigms, such as stochastic computing, have also been used for fault-tolerant image smoothing.

Wireless Communication

Wireless Sensor Networks (WSNs) have emerged as a promising candidate solution for providing ambient intelligence. WSNs include Sensor Nodes (SNs) that monitor the physical environment to provide the observations for various cyberphysical systems. Consequently, the SNs are usually deployed in harsh environments that may result in reliability issues with an SN's sensing, computation, and communication functions. Overall, faults in an WSN may arise from *node-*, *sink-*, *network-level* (Effah and Thiare 2018). The corresponding fault tolerance approaches include *centralized*, *decentralized*, and *hybrid* approaches with varying trade-offs in reliability and communication costs (Adday et al. 2022). Using traditional redundancy-based methods for computation, on SNs, gateways, and base-station, and retransmission using additional links for communication would incur large costs. A cross-layer approach to fault management in WSNs is presented by Vihman et al. (2020). The presented approach aims to eliminate any form of hardware and temporal redundancy and, instead, uses the data gathered by the various levels – SNs at the Edge, gateways at the Fog and the Cloud – to detect and isolate the faults. Since the SNs are usually resource-constrained systems that aim to minimize energy consumption, such cross-layer reliability approaches can be used to ensure high availability with minimal resource overheads resulting from duplicated execution.

Fault Tolerance in Emerging Technologies

The last two decades have witnessed exponential growth in the application areas using electronic systems. Much of this growth has been fuelled by emerging technologies – both in terms of new devices beyond Complementary Metal-Oxide Semiconductor (CMOS) and novel computing paradigms such as AI/ML, IoTs, etc. In this section, we provide an overview of reliability issues and fault tolerance across some emerging technologies.

Emerging Memory Technologies

DRAM has long been the choice substrate for architecting main memory subsystems due to its low cost per bit. However, DRAM is a fundamental performance and energy bottleneck in almost all computer systems (Wulf and McKee 1995) and is experiencing significant technology scaling challenges. DRAM-compatible emerging nonvolatile memory (NVM) technologies such as Flash, oxide-based Resistive RAM (OxRRAM), phase-change memory (PCM), and spin transfer torque or spin orbit torque magnetic RAM (STT/SOT-MRAM) can address some of

these challenges (Mutlu 2013). Apart from their use as DRAM alternatives in conventional computing, NVMs are now used in neuromorphic computing (Mead 1990), which are analog computing systems (hardware) that mimic the brain's architecture. Recent demonstrations include the use of OxRRAM (Mallik et al. 2017), PCM (Nandakumar et al. 2018), Ferroelectric RAM (FeRAM) (Mulaosmanovic et al. 2017), and STT-/SoT-MRAM (Vincent et al. 2015).

We discuss the reliability implication for NVM-based neuromorphic hardware. To this end, Fig. 19A and B show a simple feedforward and a recurrent neural network architecture, respectively. The synaptic weights between the neurons are implemented using NVMs. In Fig. 19C we illustrate the architecture of a crossbar, which is the building block of a neuromorphic hardware. A crossbar is a 2D organization of horizontal wordlines and vertical bitlines. An NVM is integrated at the intersection of a bitline and a wordline. Finally, in Fig. 20, we illustrate a many-core design where the crossbar arrays are interconnected using a time-multiplexed interconnect (Balaji et al. 2019).

For neuromorphic hardware, reliability issues may lead to logic or memory failures. A logic failure is related to the peripheral logic of a crossbar consisting of neuron circuitries. A memory failure is related to the NVM devices implementing synaptic weights. We now elaborate these issues.

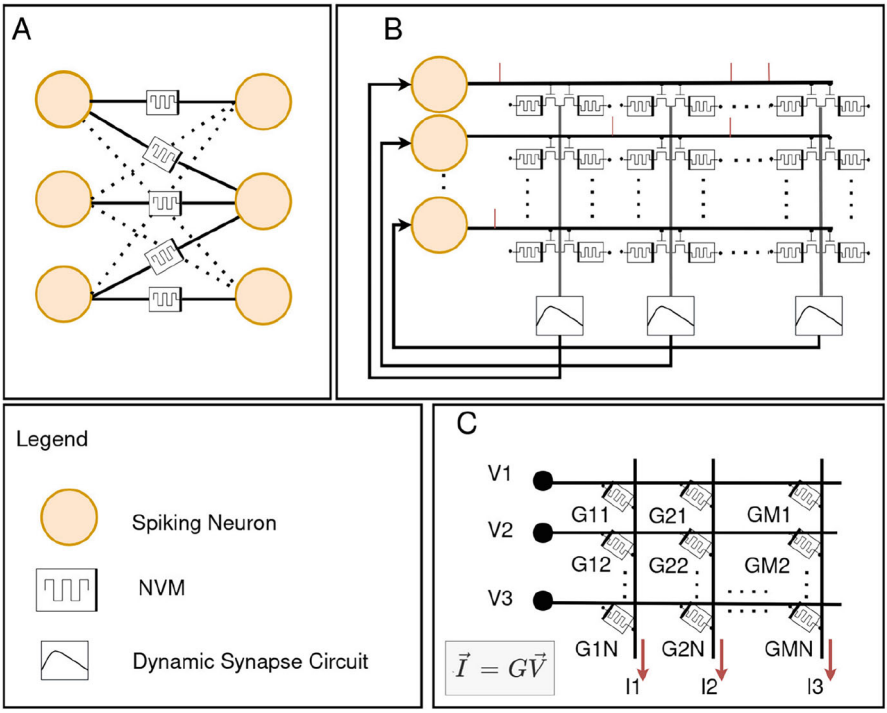


Fig. 19 Neuromorphic hardware with nonvolatile memory (NVM) synapses

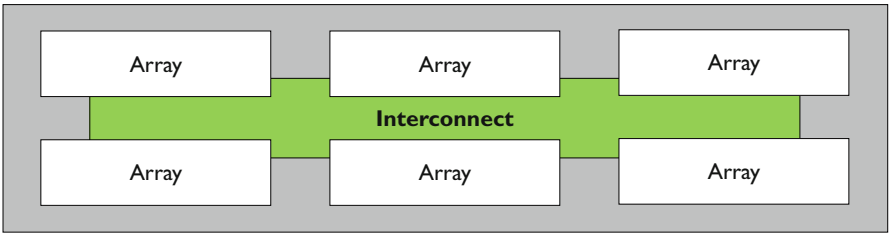


Fig. 20 A many-core neuromorphic hardware built using NVM arrays

Table 2 Reliability issues in NVMs

Reliability issues	NVMs
High voltage-related circuit aging	PCM, Flash
High current-related circuit aging	OxRAM, STT-MRAM
Read disturbance	All
Limited endurance	All

Logic Failures In a neuromorphic device, neurons are implemented using silicon circuitries. We take bias temperature instability (BTI)-induced silicon aging as an example. BTI is manifested as an increase of the threshold voltage of a transistor (Kraak et al. 2019). If neuron circuitries are used continuously, BTI aging cannot be reversed in these circuitries, resulting in hardware faults. Aggressive device scaling increases power density and temperature, which accelerates BTI aging. Apart from BTI, there are also two other dominant aging mechanisms at scaled technology nodes – TDDB and HCI. At older nodes, there are other aging mechanisms such as electromigration.

Memory Failures Memory failures in a neuromorphic device involve the breakdown of its NVMs. There are several different types of failure mechanisms reported for NVMs. Some of these failure mechanisms are common across a different NVM types, while others are NVM-specific. A thorough survey of these failure mechanisms can be found in Varshika et al. (2022). Table 2 summarizes the sources of reliability concerns in emerging NVMs.

Reliability Issues in NVMs

We provide an overview of two failure mechanisms – one specific to OxRRAM and the other to PCM.

Read Disturb Issue in OxRRAM

The resistance switching random access memory (OxRRAM) technology presents an attractive option for implementing the synaptic cells of a crossbar due to its

demonstrated potential for low-power multilevel operation and high integration density (Mallik et al. 2017). An OxRRAM cell is composed of an insulating film sandwiched between conducting electrodes forming a metal-insulator-metal (MIM) structure (see Fig. 21). Recently, filament-based metal-oxide OxRRAM implemented with transition-metal-oxides such as HfO_2, ZrO_2, and TiO_2 has received considerable attention due to their low-power and CMOS-compatible scaling (Cüppers et al. 2019).

In OxRRAM technology, the transition from HRS to one of the LRS states can occur due to a sudden decrease of the vertical filament gap on application of a stress voltage during spike propagation (Shim et al. 2020). This is illustrated in the left subfigure of Fig. 22. The transition from one of the LRS states can occur due to the lateral filament growth (Shim et al. 2020). This is illustrated in the right subfigure of Fig. 22.

Synaptic weights are represented as conductance of the insulating layer within each OxRRAM cell. To program an OxRRAM cell, elevated voltages are applied at the top and bottom electrodes, which rearranges the atomic structure of the insulating layer. Figure 21 shows the high-resistance state (HRS) and the low-resistance

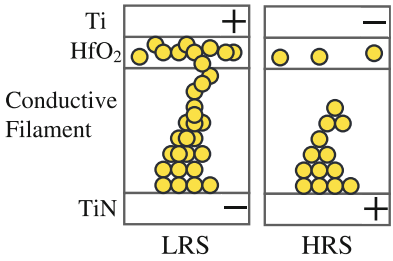


Fig. 21 Operation of an OxRRAM cell. The left subfigure shows the LRS state. The right subfigure shows the HRS state

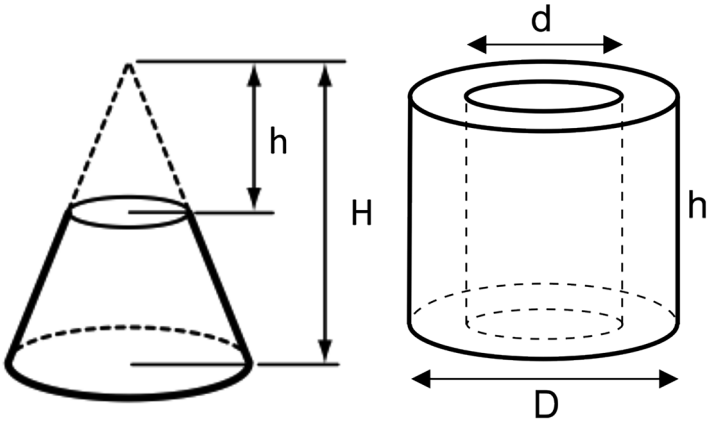


Fig. 22 Read disturbances due to structural alteration in an OxRRAM cell. The left subfigure shows a reduction of the conductive filament gap (i.e., read disturbance of HRS state) on the application of a stress voltage. The right subfigure shows the lateral growth of the conductive filament (i.e., read disturbance of LRS state) due to application of a stress voltage

state (LRS) of an OxRRAM cell. An OxRRAM cell can also be programmed into intermediate low-resistance states, allowing its multilevel operations. An OxRRAM's read disturb issue can cause the resistance state of the cell to drift away from the programmed value upon repeated access of a cell during inference. Resistance drifts can lower the inference accuracy of a machine learning model that implemented on OxRRAM-based crossbars.

Thermal Issues due to PCM's High Voltage Operations

Figure 23a illustrates how a chalcogenide semiconductor alloy is used to build a PCM cell. The amorphous phase (logic "0") in this alloy has higher resistance than its crystalline phase (logic "1"). When using only these two states, each PCM cell can implement a binary synapse. However, with precise control of the crystallization process, a PCM cell can be placed in a partially crystallized state, in which case, it can implement a multi-bit synapse. $Ge_2Sb_2Te_5$ (GST) is the most commonly used alloy for PCM due to its high amorphous-to-crystalline resistance ratio, fast switching between phases, and high endurance. However, other chalcogenide alloys are also explored due to their better data retention properties (Morikawa et al. 2007). Phase changes in a PCM cell are induced by injecting current into resistor-chalcogenide junction and heating the chalcogenide alloy to about 650°C. Figure 23b shows the different current profiles needed to program and read in a PCM device (Secco et al. 2017).

Titirsha et al. (2021) modeled the self-heating temperature of a PCM, which causes the change of phase in the cell. The self-heating temperature rises using a closed-loop feedback mechanism, which is illustrated in Fig. 24. This is described using the phenomenological model. At start of the amorphization process, the temperature of a PCM cell is equal to the ambient temperature T_{amb}. Subsequently, the PCM temperature is computed iteratively as follows. For a given crystalline fraction V_C of the GST material within the cell, the thermal conductivity k is computed using the `TC Module` and PCM resistance R_{PCM} using the `PCMR Module`. The thermal conductivity is used to compute the heat dissipation W_d

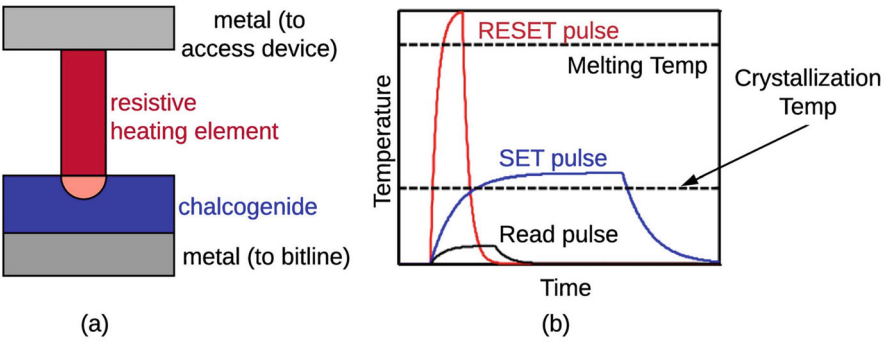


Fig. 23 (**a**) A phase-change memory (PCM) cell and (**b**) current needed to SET, RESET, and read a PCM cell

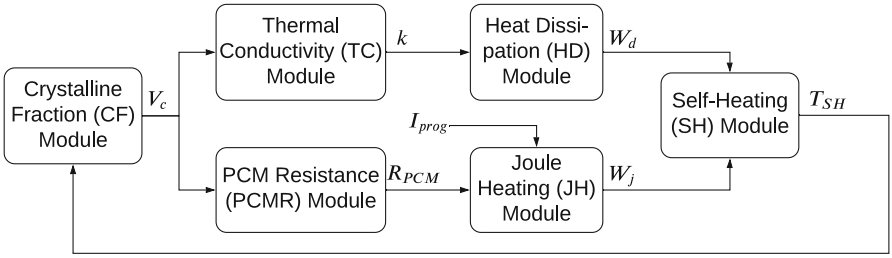


Fig. 24 Feedback-driven increase of the self-heating temperature of a PCM cell during amorphization

using the `HD Module`, while the PCM resistance is used to compute the Joule heating in the GST W_j for the programming current I_{prog} using the `JH Module`. The self-heating temperature T_{SH} is computed inside the `SH Module` using the Joule heating and the heat dissipation. Finally, the self-heating temperature is used to compute the crystallization fraction V_c using the `CF Module`. The iterative process terminates when the GST is amorphized, i.e., $V_c = 0$.

The self-heating temperature of a PCM during amorphization can lead to several thermal-related issues. First, it can lower the write endurance of the cell, where write endurance is defined as the number of times a PCM cell can be reliably programmed. In fact, Titirsha et al. (2021) have shown that endurance exponentially reduces with an increase in temperature. Second, higher PCM temperatures can lead to the aging of the CMOS circuitries around the NVM cell.

Fault Tolerance in AI/ML

Neuromorphic systems (AI/ML hardware, in general) are now used in many safety-critical applications such as monitoring vital physiological data in a healthcare application and object detection in autonomous vehicles. A fault in such a system may lead to catastrophic failure. The good news is that machine learning models are over-parameterized, which means that not neuron and synaptic failure may lead to output errors. In this section, we evaluate first evaluate the built-in error tolerance of machine learning models. Thereafter, we show how the error tolerance of a model can be improved by exploiting self-repair property of the brain.

Built-In Error Tolerance of Machine Learning Models

Deep learning models may have thousands of neurons and synapses. A key research question that researchers have tried to address is the following: **are all these neurons and synapses susceptible to errors introduced due to hardware failures?** To analyze the built-in error resilience of a model, we use the `ARES` framework to inject a random synaptic error in a model and evaluate its accuracy drop on a test set (Reagen et al. 2018). The experiment is repeated 100 times.

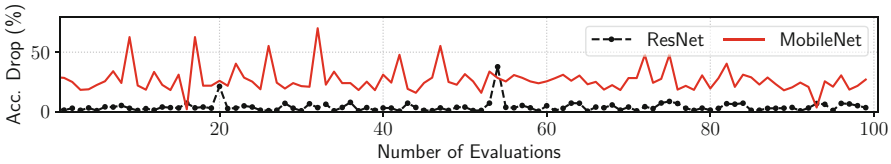


Fig. 25 Accuracy drop of ResNet and MobileNet with errors injected using the ARES framework (Reagen et al. 2018)

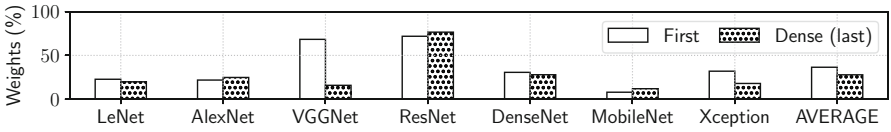


Fig. 26 Fraction of the trained weights that leads to no accuracy drop due to error injection

Figure 25 plots these results for ResNet and MobileNet, two example models. We make two key observations.

First, not all errors lead to an accuracy drop. This means that deep learning models are to a certain extent resilient to errors. Second, between ResNet and MobileNet, MobileNet has fewer synaptic weights that are error-resilient. Therefore, we see a significant accuracy drop for most errors. For ResNet, we see an accuracy drop only when an error affects a non-resilient synaptic weight. To further expand on this, Fig. 26 reports the fraction of trained synaptic weights in the first and last (dense) layers of seven models that leads to no accuracy drop when a single random error is injected.

We observe that an average 37% of synaptic weights in the first layer and 30% in the last layer are resilient to errors. Therefore, these synaptic weights do not need fault tolerance. The cost of fault tolerance can be reduced by providing solutions for only the neurons and synapses, which lead to accuracy drop when implemented on a faulty device. For instance, Liu et al. (2017) propose a methodology to rescue bit failures in NVM-based neuromorphic hardware in order to restore the computational accuracy. The design methodology consists of three steps. First, authors propose to identify weights of a machine learning model that have lower impact on accuracy. Essentially, model weights are categorized into significant and insignificant weights. Next, authors propose a retraining algorithm to compensate for single-bit failure by retuning the trainable weights. Finally, during the mapping step, a redundancy mapping scheme is used to further improve the computation accuracy.

Fault Tolerance via Self-Repair

Recently, fault tolerance properties are embedded in machine learning applications by exploiting the self-repair property of the mammalian brain. It is shown that the brain copes with damaged neurons using astrocytes, the star-shaped glial cells of

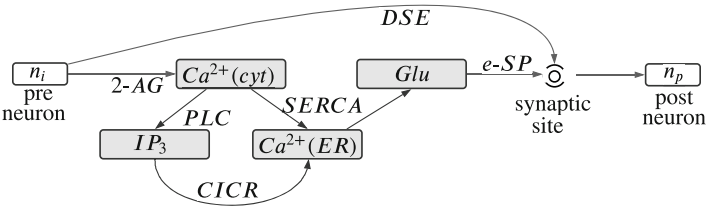


Fig. 27 Operation of an astrocyte (gray blocks)

the brain (Parpura et al. 1994). Self-repair is facilitated by restoring the spike firing frequency of a failed neuron using a closed-loop retrograde feedback signal.

Figure 27 illustrates how an astrocyte regulates the neuronal activity at a synaptic site using a closed-loop feedback mechanism.

Astrocyte causes a transient increase of intracellular calcium (Ca^{2+}) levels, which serves as the catalyst for self-repair. Ca^{2+}-induced Ca^{2+} release (CICR) is the main mechanism to regulate Ca^{2+} in the healthy brain. CICR is triggered by inosital 1,4,5-triphosphate (IP_3), which is produced upon astrocyte activation. To describe the operation of the astrocyte, let $\delta(t-\tau)$ be a spike at time τ from the neuron n_i. This spike triggers the release of 2-arachidonyl glycerol (2-AG), a type of endocannabinoid responsible for stimulating the cytosolic calcium Ca^{2+} (cyt). The quantity of 2-AG produced is governed by the ordinary differential equation (ODE)

$$\frac{dAG}{dt} = \frac{-AG}{\tau_{AG}} + r_{AG} \cdot \delta(t-\tau), \tag{2}$$

where AG is the quantity of 2-AG, τ_{AG} is the rate of decay, and r_{AG} is the rate of production of 2-AG.

On one pathway, the cytosolic calcium is absorbed by the endoplasmic reticulum (ER) via the sarcoendoplasmic reticulum Ca^{2+}-ATPase (SERCA) pumps, and on the other pathway, the cytosolic calcium enhances the phospholipase C (PLC) activation process. This event increases IP_3 production and ER intracellular calcium release via the CICR mechanism.

The intracellular astrocytic calcium dynamics control the glutamate (Glu) release from the astrocyte, which is governed by

$$\frac{dGlu}{dt} = \frac{-Glu}{\tau_{Glu}} + r_{Glu}(t - t_{Ca}), \tag{3}$$

where τ_{Glu} is the rate of decay, r_{Glu} is the rate of production of glutamate, and t_{Ca} is time at which Ca^{2+} crosses the release threshold. The glutamate generates e-SP, the indirect signal to the synaptic site. e-SP is related to Glu using the following ODE:

$$\frac{deSP}{dt} = \frac{-eSP}{\tau_{eSP}} + \frac{m_{eSP}}{\tau_{eSP}} Glu(t), \tag{4}$$

where τ_{eSP} is the decay rate of e-SP and m_{eSP} is a scaling factor.

Finally, there exists a direct signaling pathway (DSE) from neuron n_i to the synaptic site. The DSE is given by

$$DSE = -K_{AG} \cdot AG(t), \tag{5}$$

where K_{AG} is a constant. Overall, the synaptic transmission probability (PR) at the synaptic site is

$$PR(t) = PR(0) + PR(0)\left(\frac{DSE(t) + eSP(t)}{100}\right) \tag{6}$$

In the brain, each astrocyte encloses multiple synapses connected to a neuron. Figure 28a shows an original network of neurons, while Fig. 28b shows these neurons enclosed using an astrocyte.

To understand the self-repair mechanism, consider neuron n_i in Fig. 27 fails to fire a spike. This can happen due to (1) a logic fault, which may impact the neuron circuitry implementing n_i, or (2) a memory fault, which may impact the presynaptic connections of n_i. Without the astrocyte, the spike firing rate at the synaptic site would decrease. However, because of the astrocyte, 2-AG production reduces (Eq. 2). This increases the DSE (Eq. 5). Therefore, the PR increases (Eq. 6) along with an increase of the spike firing frequency at the synaptic site.

To provide more insight into this error-resilient behavior, Fig. 29 illustrates the operation of four input leaky-integrate-and-fire (LIF) neurons connected to an output LIF neuron via four synapses. Input spikes raise the membrane voltage of the output neuron, which fires an output spike when its membrane voltage crosses a threshold. We illustrate the impact of a change in the synaptic weight w_4, connecting input i_4 with output o. In this example, we show three quantized states of the synaptic weight w_4 — w_x, w_y, and w_z. It can be observed that an output spike is generated for all three quantized states of w_4 (shown as outputs o_x, o_y, and o_z). This means that the synaptic weight w_4 is tolerant to memory failures. On the other hand, an output spike is generated even when the input neuron i_3 fails to generate a spike. The corresponding membrane voltage is shown with the dashed line and the output as o_r in the figure. This means that the LIF neuron i_3 is tolerant to logic failures.

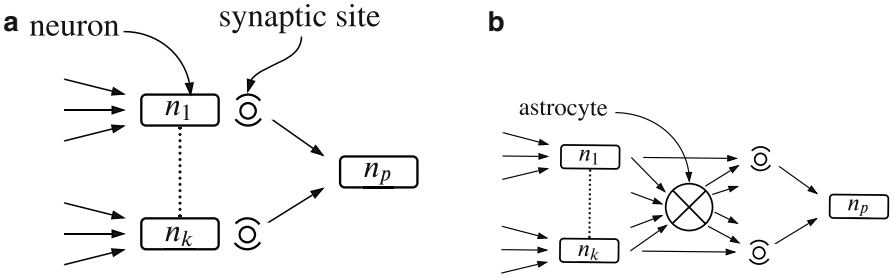


Fig. 28 Inserting an astrocyte in a neural network. (**a**) Original network of neurons and synapses. (**b**) Astrocyte-modulated neurons and synapses

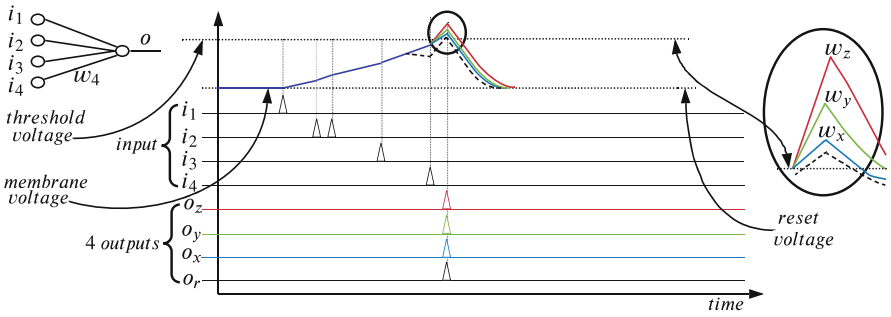


Fig. 29 Built-in error tolerance of a model

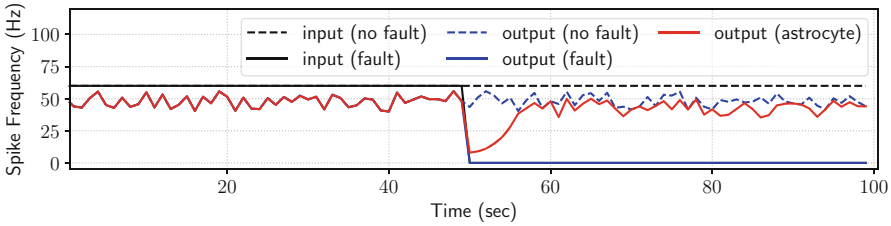


Fig. 30 Self-repair mechanism of an astrocyte

Figure 30 illustrates the self-repair mechanism. The input neuron n_i is excited with Poisson spike events having a mean spike rate of 60 Hz. The input is interrupted at around 50 sec. It can be observed that the firing frequency at the synaptic site connected to n_i drops to 0. This is indicated with the label *output (fault)*. Using astrocyte, the firing frequency can be restored partially as illustrated using the label *output (astrocyte)*. The **frequency reconstruction error** of an astrocyte is defined as the difference between the recovered and ideal spike frequency at a faulty synaptic site. The **degree of fault tolerance** (also called degree of self-recovery) is related to the inverse of this frequency reconstruction error.

To provide fault tolerance, astrrocytes need to be inserted into a machine learning model, such that its neurons and synapses can be protected from logic and memory failures when the model is implemented on a neuromorphic hardware. Recent works, Isik et al. (2022), have shown that astrocyte-based neural networks can facilitate a low-cost fault tolerance solution compared to many conventional fault-tolerant implementations – software solutions such as model replication (Latifi et al. 2020) and error prediction coding (Park et al. 2020) and hardware solutions such as approximation (Siddique et al. 2021) and redundant mapping (Yuan et al. 2021).

Conclusion

The increasing prevalence of AI/ML-based systems and the quest for *smart things* are expected to drive the growth and innovations in electronic systems for the next few decades. As we move from isolated smart things toward collective intelligence, electronic systems are being deployed across varying operating environments. Consequently, the reliability requirements are going to vary across not just application domains but also on each system's operating environment. Therefore, designing fault-tolerant architectures forms an indispensable component of application-specific system design. While the fundamental approaches to fault tolerance, such as *redundancy* and *diversity*, still apply, these approaches need to be tailored at the system-level for emerging technologies. To this end, this chapter provides an overview of both the essential background and the emerging approaches to designing fault-tolerant architectures. Additionally, the chapter also provides a brief overview of state-of-the-art architecture-level methods for fault-tolerant processing.

The scope for related future research primarily includes the design of application-specific fault tolerance. Novel approaches such as cross-layer reliability and information processing factory (Rambo et al. 2019) provide a path for implementing cost-efficient, dynamically adaptable fault tolerance. Further, the exploration into leveraging the benefits of newer computing paradigms such as approximate computing and memory-oriented computing for inherently robust applications can result in fault tolerance with reduced overheads.

Glossary

ACE Architecturally Correct Execution.
AVF Architectural Vulnerability Factor.
BErR Backward Error Recovery.
BISR Built-in Self Recovery.
BIST Built-in Self Test.
BTI Bias Temperature Instability.
CGRA Coarse-Grained Reconfigurable Arrays.
CLR Cross-layer Reliability.
CMOS Complementary Metal-Oxide Semiconductor.
CNN Convolutional Neural Networks.
COTS commercial-off-the-shelf.
DCH Divide and Conquer Hamming.
DEC Double-bit-Error-Correcting.
DED Double-bit-Error-Detecting.
DIMM Dual in-line-Memory Module.
DMR Dual Modular Redundancy.
DND Double-Nibble-error-Detecting.
DPR Dynamic Partial Reconfiguration.
DRAM Dynamic Random Access Memory.

DSE Design Space Exploration.
DSP Digital Signal Processing.
DVS Dynamic Voltage Scaling.
ECC Error Checking and Correcting.
EM Electromigration.
FErR Forward Error Recovery.
FI Fault Injection.
FPGA Field-Programmable Gate Array.
FVI Function Vulnerability Index.
HCI Hot Carrier Injection.
HDD Hard Disk Drive.
HDL Hardware Description Language.
IC integrated circuits.
IVI Instruction Vulnerability Index.
L1 Level 1.
LLC Last Level Cache.
LSEs Latent Section Errors.
MAC Multiply and Accumulate.
MOS Metal-Oxide Semiconductor.
MTBF Mean Time between Failures.
MTTC Mean Time To Crash.
MTTD Mean Time To Detection.
MTTDL Mean Time to Data Loss.
MTTF Mean Time To Failure.
NMR *N*-Modular Redundancy.
NoC Network-on-Chip.
OSI Open Systems Interconnection.
PE Processing Element.
PRR Partially Reconfigurable Region.
PTM Probabilistic Transfer Matrix.
QoS Quality of Service.
RAID Redundant Array of Inexpensive Disks.
RTL Register Transfer Level.
SEC Single-bit-Error-Correcting.
SEL Single Event Latchup.
SER Soft Error Rate.
SEU Single Event Upset.
SMT Simultaneous Multi-threading.
SN Sensor Node.
SNC Single-Nibble-error-Correcting.
SODIMM Small Outline DIMM.
SRAM Static Random Access Memory.
SRMT Software-based Redundant Multi-threading.
SRT Simultaneous and Redundant Threading.
SSD Solid-state Drive.

TDDB Time Dependent Dielectric Breakdown.
TED Triple-bit-Error-Detecting.
TID Total Ionizing Dose.
TMR Triple Modular Redundancy.
UDEs Undetected Disk Errors.
WCET Worst-Case Executing Time.
WSN Wireless Sensor Network.

References

Adday GH, Subramaniam SK, Zukarnain ZA, Samian N (2022) Fault tolerance structures in wireless sensor networks (wsns): survey, classification, and future directions. Sensors 22(16). https://doi.org/10.3390/s22166041, https://www.mdpi.com/1424-8220/22/16/6041

Austin TM (1999) Diva: a reliable substrate for deep submicron microarchitecture design. In: MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture, pp 196–207. https://doi.org/10.1109/MICRO.1999.809458

Avizienis A, Laprie J, Randell B, Landwehr C (2004) Basic concepts and taxonomy of dependable and secure computing. IEEE Trans Depend Secure Comput 1(1):11–33. https://doi.org/10.1109/TDSC.2004.2

Arzt E, Kraft O, Sanchez JE, Bader S, Nix WD (1992) Electromigration resistance and mechanical strength

Balaji A, Wu Y, Das A, Catthoor F, Schaafsma S (2019) Exploration of segmented bus as scalable global interconnect for neuromorphic computing. In: GLSVLSI

Bar-El H, Choukri H, Naccache D, Tunstall M, Whelan C (2006) The sorcerer's apprentice guide to fault attacks. Proc IEEE 94(2):370–382. https://doi.org/10.1109/JPROC.2005.862424

Baraza J, Gracia J, Gil D, Gil P (2002) A prototype of a vhdl-based fault injection tool: description and application. J. Syst. Architect 47(10):847–867. https://doi.org/https://doi.org/10.1016/S1383-7621(01)00036-4, https://www.sciencedirect.com/science/article/pii/S1383762101000364

Biasielli M, Bolchini C, Cassano L, Mazzeo A, Miele A (2022) Approximation-based fault tolerance in image processing applications. IEEE Trans Emerg Top Comput 10(2):648–661. https://doi.org/10.1109/TETC.2021.3100623

Binder D, Smith EC, Holman AB (1975) Satellite anomalies from galactic cosmic rays. IEEE Trans Nucl Sci 22(6):2675–2680. https://doi.org/10.1109/TNS.1975.4328188

Blaauw D, Kalaiselvan S, Lai K, Ma W, Pant S, Tokunaga C, Das S, Bull D (2008) Razor II: in situ error detection and correction for PVT and SER tolerance. In: 2008 IEEE International Solid-State Circuits Conference – Digest of Technical Papers, pp 400–622. https://doi.org/10.1109/ISSCC.2008.4523226

Carter NP, Naeimi H, Gardner DS (2010) Design techniques for cross-layer resilience. In: 2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010), pp 1023–1028. https://doi.org/10.1109/DATE.2010.5456960

Chen PM, Lee EK, Gibson GA, Katz RH, Patterson DA (1994) Raid: high-performance, reliable secondary storage. ACM Comput Surv 26(2):145–185. https://doi.org/10.1145/176979.176981

Cheng E, Mirkhani S, Szafaryn LG, Cher CY, Cho H, Skadron K, Stan MR, Lilja K, Abraham JA, Bose P, Mitra S (2016) Clear: cross-layer exploration for architecting resilience – combining hardware and software techniques to tolerate soft errors in processor cores. In: Proceedings of the 53rd Annual Design Automation Conference, DAC'16. ACM, New York, pp 68:1–68:6. https://doi.org/10.1145/2897937.2897996, http://doi.acm.org/10.1145/2897937.2897996

Cho H, Cheng E, Shepherd T, Cher CY, Mitra S (2017) System-level effects of soft errors in uncore components. IEEE Trans Comput-Aided Design Integr Circuits Syst 36(9):1497–1510. https://doi.org/10.1109/TCAD.2017.2651824

Cüppers F, Menzel S, Bengel C, Hardtdegen A, Von Witzleben M, Böttger U, Waser R, Hoffmann-Eifert S (2019) Exploiting the switching dynamics of HfO2-based ReRAM devices for reliable analog memristive behavior. APL Mater 7(9):091105. https://doi.org/10.1063/1.5108654

Das A, Kumar A, Veeravalli B (2013) Reliability-driven task mapping for lifetime extension of networks-on-chip based multiprocessor systems. In: 2013 Design, Automation Test in Europe Conference Exhibition (DATE), pp 689–694. https://doi.org/10.7873/DATE.2013.149

Dubrova E (2013) Introduction. Springer, New York, pp 1–4. https://doi.org/10.1007/978-1-4614-2113-9_1

Dumitriu V, Kirischian L, Kirischian V (2016) Run-time recovery mechanism for transient and permanent hardware faults based on distributed, self-organized dynamic partially reconfigurable systems. IEEE Trans Comput 65(9):2835–2847. https://doi.org/10.1109/TC.2015.2506558

Effah E, Thiare O (2018) Survey: faults, fault detection and fault tolerance techniques in wireless sensor networks. Int J Comput Sci Inf Secur(IJCSIS) 16(10):1–14

Hamming RW (1950) Error detecting and error correcting codes. Bell Syst Tech J 29(2):147–160. https://doi.org/10.1002/j.1538-7305.1950.tb00463.x

Henkel J, Bauer L, Zhang H, Rehman S, Shafique M (2014) Multi-layer dependability: From microarchitecture to application level. In: Proceedings of the 51st Annual Design Automation Conference. Association for Computing Machinery, New York, p 1–6. https://doi.org/10.1145/2593069.2596683

Hsiao MY (1970) A class of optimal minimum odd-weight-column sec-ded codes. IBM J Res Develop 14(4):395–401. https://doi.org/10.1147/rd.144.0395

Isik M, Paul A, Varshika ML, Das A (2022) A design methodology for fault-tolerant computing using astrocyte neural networks. In: Proceedings of the 19th ACM International Conference on Computing Frontiers, pp 169–172

Kakoee MR, Bertacco V, Benini L (2011) Relinoc: a reliable network for priority-based on-chip communication. In: 2011 Design, Automation Test in Europe, pp 1–6. https://doi.org/10.1109/DATE.2011.5763112

Karaklajić D, Schmidt JM, Verbauwhede I (2013) Hardware designer's guide to fault attacks. IEEE Trans Very Large Scale Integr (VLSI) Syst 21(12):2295–2306. https://doi.org/10.1109/TVLSI.2012.2231707

Kim J, Sullivan M, Erez M (2015) Bamboo ECC: strong, safe, and flexible codes for reliable computer memory. In: 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), pp 101–112. https://doi.org/10.1109/HPCA.2015.7056025

Kim BS, Choi J, Min SL (2019) Design tradeoffs for ssd reliability. In: Proceedings of the 17th USENIX Conference on File and Storage Technologies, FAST'19. USENIX Association, USA, pp 281–294

Koch D, Haubelt C, Teich J (2007) Efficient hardware checkpointing: concepts, overhead analysis, and implementation. In: Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays, FPGA'07. ACM, New York, pp 188–196. https://doi.org/10.1145/1216919.1216950, http://doi.acm.org/10.1145/1216919.1216950

Kraak D, Taouil M, Agbo I, Hamdioui S, Weckx P, Cosemans S, Catthoor F (2019) Parametric and Functional Degradation Analysis of Complete 14-nm FinFET SRAM. TVLSI. https://doi.org/10.1109/TVLSI.2019.2902881

Kriebel F, Rehman S, Sun D, Shafique M, Henkel J (2014) Aser: adaptive soft error resilience for reliability-heterogeneous processors in the dark silicon era. In: 2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC), pp 1–6. https://doi.org/10.1145/2593069.2593094

Krishnaswamy S, Viamontes GF, Markov IL, Hayes JP (2008) Probabilistic transfer matrices in symbolic reliability analysis of logic circuits. ACM Trans Des Autom Electron Syst 13(1). https://doi.org/10.1145/1297666.1297674

Latifi S, Zamirai B, Mahlke S (2020) PolygraphMR: enhancing the reliability and dependability of CNNs. In: DSN

Liu C, Hu M, Strachan JP, Li H (2017) Rescuing memristor-based neuromorphic design with high defects. In: DAC

Mallik A, Garbin D, Fantini A, Rodopoulos D, Degraeve R, Stuijt J, Das A, Schaafsma S, Debacker P, Donadio G et al (2017) Design-technology co-optimization for OxRRAM-based synaptic processing unit. In: VLSIT

Mead C (1990) Neuromorphic electronic systems. Proc IEEE, vol. 78(10), pp. 1629–1636. https://doi.org/10.1109/5.58356

Mohanram K, Touba NA (2003) Cost-effective approach for reducing soft error failure rate in logic circuits. In: International Test Conference, 2003. Proceedings, vol 1, ITC 2003, pp 893–901. https://doi.org/10.1109/TEST.2003.1271075

Moore GE (2006) Cramming more components onto integrated circuits, reprinted from electronics, vol 38, number 8, 19 Apr, 1965, pp.114 ff. IEEE Solid-State Circuits Soc Newslett 11(3):33–35. https://doi.org/10.1109/N-SSC.2006.4785860

Morikawa T, Kurotsuchi K, Kinoshita M, Matsuzaki N, Matsui Y, Fujisaki Y, Hanzawa S, Kotabe A, Terao M, Moriya H, et al. (2007) Doped In-Ge-Te phase change memory featuring stable operation and good data retention. In: IEDM

Mukherjee SS, Kontz M, Reinhardt SK (2002) Detailed design and evaluation of redundant multithreading alternatives. In: Proceedings 29th Annual International Symposium on Computer Architecture, pp 99–110. https://doi.org/10.1109/ISCA.2002.1003566

Mukherjee SS, Weaver C, Emer J, Reinhardt SK, Austin T (2003) A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In: Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36, pp 29–40. https://doi.org/10.1109/MICRO.2003.1253181

Mulaosmanovic H, Ocker J, Müller S, Noack M, Müller J, Polakowski P, Mikolajick T, Slesazeck S (2017) Novel ferroelectric FET based synapse for neuromorphic systems. In: VLSIT

Mutlu O (2013) Memory scaling: a systems architecture perspective. In: IMW

Nandakumar SR, Le Gallo M, Boybat I, Rajendran B, Sebastian A, Eleftheriou E (2018) A phase-change memory model for neuromorphic computing. JAP 124(15): 152135. https://doi.org/10.1063/1.5042408

Park S, Li S, Zhang Z, Mahlke S (2020) Low-cost prediction-based fault protection strategy. In: CGO

Parpura V, Basarsky TA, Liu F, Jeftinija K, Jeftinija S, Haydon PG (1994) Glutamate-mediated astrocyte–neuron signalling. Nature 369(6483), 744–747. https://doi.org/10.1038/369744a0

Patterson DA, Gibson G, Katz RH (1988) A case for redundant arrays of inexpensive disks (raid). SIGMOD Rec 17(3):109–116. https://doi.org/10.1145/971701.50214

Postman J, Chiang P (2012) A survey addressing on-chip interconnect: energy and reliability considerations. ISRN Electronics 2012

Rambo EA, Kadeed T, Ernst R, Seo M, Kurdahi F, Donyanavard B, de Melo CB, Maity B, Moazzemi K, Stewart K, Yi S, Rahmani AM, Dutt N, Maurer F, Vu Doan NA, Surhonne A, Wild T, Herkersdorf A (2019) The information processing factory: A paradigm for life cycle management of dependable systems. In: 2019 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), pp 1–10. https://doi.org/10.1145/3349567.3357391

Rao TRN (1974) Error Coding for Arithmetic Processors. Academic Press, Inc., Orlando

Rao RR, Blaauw D, Sylvester D (2006) Soft error reduction in combinational logic using gate resizing and flipflop selection. In: 2006 IEEE/ACM International Conference on Computer Aided Design, pp 502–509. https://doi.org/10.1109/ICCAD.2006.320165

Reagen B, Gupta U, Pentecost L, Whatmough P, Lee SK, Mulholland N, Brooks D, Wei GY (2018) Ares: a framework for quantifying the resilience of deep neural networks. In: DAC

Reed I, Solomon G (1960) Polynomial codes over certain finite fields. J Soc Ind Appl Math 8(2):300–304. https://doi.org/10.1137/0108018

Rehman S, Chen K, Kriebel F, Toma A, Shafique M, Chen J, Henkel J (2016) Cross-layer software dependability on unreliable hardware. IEEE Trans Comput 65(1):80–94. https://doi.org/10.1109/TC.2015.2417554

Sahoo SS (2019) A cross-layer reliability-integrated system-level design methodology for heterogeneous multiprocessor SoC-based embedded systems. PhD thesis, National University of Singapore (Singapore)

Sahoo SS, Veeravalli B, Kumar A (2016) Cross-layer fault-tolerant design of real-time systems. In: DFTS, pp 63–68. https://doi.org/10.1109/DFT.2016.7684071

Sahoo SS, Nguyen TDA, Veeravalli B, Kumar A (2018a) Lifetime-aware design methodology for dynamic partially reconfigurable systems. In: 2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC), pp 393–398. https://doi.org/10.1109/ASPDAC.2018.8297355

Sahoo SS, Veeravalli B, Kumar A (2018b) CLRFrame: an analysis framework for designing cross-layer reliability in embedded systems. In: 31st International Conference on VLSI Design and 17th International Conference on Embedded Systems, VLSID 2018, Pune, India, 6–10 Jan, 2018, pp 307–312. https://doi.org/10.1109/VLSID.2018.81, http://doi.ieeecomputersociety.org/10.1109/VLSID.2018.81

Sahoo S, Nguyen T, Veeravalli B, Kumar A (2019) Multi-objective design space exploration for system partitioning of fpga-based dynamic partially reconfigurable systems. Integration 67:95–107. https://doi.org/10.1016/j.vlsi.2018.10.006

Sahoo SS, Veeravalli B, Kumar A (2020a) CL(R)Early: an Early-stage DSE Methodology for Cross-Layer Reliability-aware Heterogeneous Embedded Systems. In: 2020 57th ACM/IEEE Design Automation Conference (DAC), pp 1–6. https://doi.org/10.1109/DAC18072.2020.9218747

Sahoo SS, Veeravalli B, Kumar A (2020b) Markov chain-based modeling and analysis of checkpointing with rollback recovery for efficient dse in soft real-time systems. In: 2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), pp 1–6. https://doi.org/10.1109/DFT50435.2020.9250892

Santini T, Rech P, Sartor A, Corrêa UB, Carro L, Wagner F (2015) Evaluation of failures masking across the software stack. MEDIAN

Santos R, Venkataraman S, Kumar A (2017) Scrubbing mechanism for heterogeneous applications in reconfigurable devices. ACM Trans Des Autom Electron Syst 22(2). https://doi.org/10.1145/2997646

Schmidt AG, French M (2013) Fast lossless image compression with radiation hardening by hardware/software co-design on platform fpgas. In: 2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors, pp 103–106. https://doi.org/10.1109/ASAP.2013.6567560

Secco J, Corinto F, Sebastian A (2017) Flux–charge memristor model for phase change memory. TCAS II: Express Briefs

Shafique M, Rehman S, Aceituno PV, Henkel J (2013) Exploiting program-level masking and error propagation for constrained reliability optimization. In: 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC), pp 1–9. https://doi.org/10.1145/2463209.2488755

Shim B, Shanbhag N (2006) Energy-efficient soft error-tolerant digital signal processing. IEEE Trans Very Large Scale Integr (VLSI) Syst 14(4):336–348. https://doi.org/10.1109/TVLSI.2006.874359

Shim B, Sridhara S, Shanbhag N (2004) Reliable low-power digital signal processing via reduced precision redundancy. IEEE Trans Very Large Scale Integr (VLSI) Syst 12(5):497–510. https://doi.org/10.1109/TVLSI.2004.826201

Shim W, Luo Y, Seo Js, Yu S (2020) Impact of read disturb on multilevel RRAM based inference engine: experiments and model prediction. In: IRPS

Shubu M (2008) Architecture design for soft errors. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA

Siddique A, Basu K, Hoque KA (2021) Exploring fault-energy trade-offs in approximate DNN hardware accelerators. In: ISQED

Slayman CW (2005) Cache and memory error detection, correction, and reduction techniques for terrestrial servers and workstations. IEEE Trans Device Mater Reliab 5(3):397–404. https://doi.org/10.1109/TDMR.2005.856487

Slegel TJ, Averill RM, Check MA, Giamei BC, Krumm BW, Krygowski CA, Li WH, Liptay JS, MacDougall JD, McPherson TJ, Navarro JA, Schwarz EM, Shum K, Webb CF (1999) Ibm's S/390 G5 microprocessor design. IEEE Micro 19(2):12–23. https://doi.org/10.1109/40.755464

Sorin DJ (2009) Fault tolerant computer architecture. Syn Lectures Comput Architect 4(1):1–104

Srinivasan S, Krishnan R, Mangalagiri P, Xie Y, Narayanan V, Irwin MJ, Sarpatwari K (2008) Toward increasing fpga lifetime. IEEE Trans Depend Secure Comput 5(2):115–127. https://doi.org/10.1109/TDSC.2007.70235

Titirsha T, Song S, Das A, Krichmar J, Dutt N, Kandasamy N, Catthoor F (2022) Endurance-aware mapping of spiking neural networks to neuromorphic hardware. TPDS 33(2):288–301. https://doi.org/10.1109/TPDS.2021.3065591

Varshika ML, Corradi F, Das A (2022) Nonvolatile memories in spiking neural network architectures: current and emerging trends. Electronics 11(10):1610. https://doi.org/10.3390/electronics11101610

Vihman L, Kruusmaa M, Raik J (2020) Data-driven cross-layer fault management architecture for sensor networks. In: 2020 16th European Dependable Computing Conference (EDCC), pp 33–40. https://doi.org/10.1109/EDCC51268.2020.00015

Vincent AF, Larroque J, Locatelli N, Romdhane NB, Bichler O, Gamrat C, Zhao WS, Klein JO, Galdin-Retailleau S, Querlioz D (2015) Spin-transfer torque magnetic memory as a stochastic memristive synapse for neuromorphic systems. TBCAS 9(2):166–174. https://doi.org/10.1109/TBCAS.2015.2414423

Wang Z, Chattopadhyay A (2017) High-level Estimation and Exploration of Reliability for Multi-processor System-on-chip. Springer. https://link.springer.com/book/10.1007/978-981-10-1073-6

Wang Z, Li R, Chattopadhyay A (2013) Opportunistic redundancy for improving reliability of embedded processors. In: 2013 8th IEEE Design and Test Symposium, pp 1–6. https://doi.org/10.1109/IDT.2013.6727090

Wang Z, Paul G, Chattopadhyay A (2014) Processor design with asymmetric reliability. In: 2014 IEEE Computer Society Annual Symposium on VLSI, pp 565–570. https://doi.org/10.1109/ISVLSI.2014.63

Wang Z, Karakonstantis G, Chattopadhyay A (2016) A low overhead error confinement method based on application statistical characteristics. In: 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp 1168–1171

Wirthlin MJ, Keller AM, McCloskey C, Ridd P, Lee D, Draper J (2016) SEU mitigation and validation of the LEON3 soft processor using triple modular redundancy for space processing. In: Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '16. ACM, New York, pp 205–214. https://doi.org/10.1145/2847263.2847278, http://doi.acm.org/10.1145/2847263.2847278

Wulf WA, McKee SA (1995) Hitting the memory wall: implications of the obvious. SIGARCH Comput. Archit. News 23(1):20–24. https://doi.org/10.1145/216585.216588

Xiang Y, Chantem T, Dick RP, Hu XS, Shang L (2010) System-level reliability modeling for mpsocs. In: 2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), pp 297–306

Yoon DH, Erez M (2010) Virtualized and flexible ecc for main memory. In: Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems, ASPLOS XV. ACM, New York, pp 397–408. https://doi.org/10.1145/1736020.1736064, http://doi.acm.org/10.1145/1736020.1736064

Yuan G, Liao Z, Ma X, Cai Y, Kong Z, Shen X, Fu J, Li Z, Zhang C, Peng H, et al. (2021) Improving DNN fault tolerance using weight pruning and differential crossbar mapping for ReRAM-based edge AI. In: ISQED

Architectures for Machine Learning

10

Yongkui Yang, Chao Chen, and Zheng Wang

Contents

Introduction . . . 322
Architectures for Neuromorphic Computing . . . 324
Biological Computing Models and Learning Methods . . . 324
Microarchitecture for Neuromorphic Computing . . . 330
Circuit-Level Design Considerations . . . 335
Prominent Neuromorphic Chips . . . 343
SpiNNaker . . . 343
Neurogrid . . . 344
BrainScales . . . 345
LaCSNN . . . 345
TrueNorth . . . 346
Loihi . . . 347
ODIN . . . 348
Tianjic . . . 348
Architectures for Artificial Neural Networks . . . 349
Design Metrics for ANN Architectures . . . 350
Design Abstractions and Trade-Offs . . . 355
Selective ANN Architectures and Circuits . . . 357
Architectures for Classic Machine Learning . . . 370
Conclusions . . . 371
References . . . 372

Abstract

The term "artificial intelligence (AI)" was coined in 1956, and its development has undergone periods of extreme hype and periods of strong disillusionment since then. Today, AI has received tremendous attention from both academia

Y. Yang · C. Chen · Z. Wang (✉)
Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, Shenzhen, China
e-mail: yk.yang@siat.ac.cn; chao.chen@siat.ac.cn; zheng.wang@siat.ac.cn

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_12

and industry, and it will remain one of the hottest topics in the foreseeable future. A subset of AI named machine learning (ML) has achieved great success throughout a huge variety of fields, such as computer vision, natural language processing, and computer gaming. ML was first proposed to endow machine the ability to imitate the learning process of the human brain using neuromorphic models. However, the modelling complexity and limited computing capabilities of machines hindered the development of ML in its early days. Benefiting from the ever-growing computing power and availability of digital data, ML has adopted both bio-inspired spiking neural network (SNN), or neuromorphic computing, and practical artificial neural network (ANN), which have become two of the top trending methods with outstanding results.

This chapter gives a brief overview of the state-of-the-art architectures and circuits for ML. On the one hand, neuromorphic computing architectures and accelerators are investigated, including bio-inspired computational models and learning methods, microarchitecture, circuit-level design considerations, and prominent neuromorphic chips. On the other hand, architectures for ANNs are outlined, including essential design metrics on ANN accelerators and various state-of-the-art ANN architectures and circuits.

Keywords

Machine learning · Neuromorphic computing · Spiking neural network · Artificial neural network · Computer architecture · VLSI · Domain-specific computing

Introduction

Artificial intelligence (AI) is by far and will remain one of the hottest topics for mankind. However, the enthusiasm of AI research was once significantly decreased before the millennium, which is known as the "AI winter." Despite many reasons for such pessimism, the increased gap between the successful mathematical methods and the difficulty of their deployment in the infrastructures was one of the root causes. Particularly, AI methods generally thirst for huge computing power, which was unable to be met in those days. Fortunately, interest and investment in AI boomed again in the first decade of the twenty-first century, which largely attributed to the advancement of semiconductor technology and computer architectures. The intrinsic architectural parallelism and increased operating frequency in modern computers met the high demands on the computing power of the previously invented AI methods, especially for a subset of AI named machine learning.

Machine learning is referred to as "the study of computer algorithms that improve automatically through experience" (Mitchell 1997), which was initially proposed

to let the machine imitate the learning procedure of the human brain. Therefore, neuroscientists pioneered in the research of the connectivity and the structural and functional organization of the cerebrum and reached the consensus that neuron is the primitive element of the brain and the signals transmitted among neurons are temporally discrete electrical potentials named as "spike." This has given the intuition that machines with learnability should be best designed following the brain's working principle, which opens the research field of bio-inspired or neuromorphic computing. The first part of this chapter introduces the bio-inspired computational models and learning methods, followed by their architectural and circuit-level design considerations, and surveys prominent neuromorphic chips.

Nonetheless, the current biological neuron model is not yet precise, and the topology of neurons' interconnection is still under investigation. Computer scientists proposed simplified neuron models and networks, namely, artificial neural network (ANN), to solve application-oriented problems and have achieved tremendous success in computer vision, natural language processing, and computer gaming. Instead of asynchronous spike signals in neuromorphic computing, artificial neurons in ANN process real values synchronously. After the year 2010, ANN has aggressively grown larger and deeper with diverse computing patterns, which is also known as "deep learning." Admitting that ANN offers an alternative but much practical solution to endow machine intelligence, therefore, its applications are prevalent from embedded devices to high-end servers. However, conventional central processing units (CPU) are incapable to accelerate ANN's huge and parallel structure, whereas the graphical processing units (GPU) demonstrate poor energy efficiency and hence uneconomical. To address this, customized ANN accelerators have been prototyped from the year 2014 and since then heavily researched. Ventures such as NVIDIA, Intel, and Google have all fabricated their ANN accelerators for both inference and training and built various services on top. The second part of this chapter aims to provide essential design metrics on ANN accelerators and surveys on state-of-the-art ANN architectures and circuits.

Neuromorphic and ANN computing are far from defining the vast scope of architectures for machine learning. Classic machine learning algorithms such as support vector machine (SVM), and K-means, principal component analysis (PCA) are widely adopted as functional kernels executing on CPU, whereas dedicated architectures for those are relatively less explored. Due to the large volume of published works, it is impractical to collect all designs and achieve absolute exactness throughout this chapter. Nevertheless, we intend to provide a quick guide for the readers and target continuous updates. As illustrated in Fig. 1, we categorize machine learning architectures based on their implementing algorithms into bio-inspired SNN, practical ANN, and classic ML designs. In this chapter, we first illustrate SNN computing models and architectures. Afterwards, we explain design metrics and present selective architectures for ANN. A brief overview on the designs for classic machine learning is also discussed.

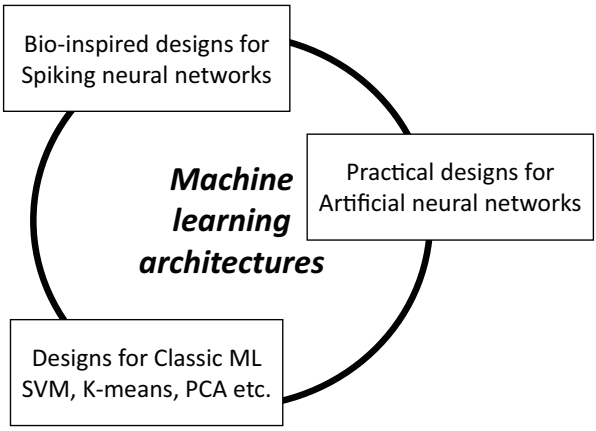


Fig. 1 Classification of Machine Learning Architectures

Architectures for Neuromorphic Computing

In this section, we review the design opportunities of current neuromorphic computing or spiking neural networks (SNN) accelerator, in terms of algorithm, microarchitecture, and circuits as shown in Fig. 2. The neuron model, synapse model, and learning method are the three most important components when designing SNN algorithms. From the aspect of neuromorphic chip's microarchitecture, neuromorphic cores consisting of neurons and synapses are in charge of computing. The communication between neuromorphic cores is normally realized by the network on chip; thanks to its ability of scalability. For the implementation, digital circuits, analog circuits, and even memristor-based circuits all are able to implement the neuromorphic chip.

Biological Computing Models and Learning Methods

In the brain, a neuron is an electrically excitable cell that normally consists of a soma (cell body), dendrites, and an axon. The synapses are the contact points where neurons communicate with each other, i.e., one neuron passes an electrical or chemical signal to another neuron. Neurons receive these electrical or chemical signals from synapses, which will change the neuron's membrane potential and will be integrated over time. When the neuron's membrane potential reaches some conditions, such as reaching a threshold, the neuron will generate an event or a spike. Figure 3 shows the biological neuron and synapse. In the human brain, each neuron has on average 7,000 synaptic connections to other neurons (Herculano-Houzel 2009).

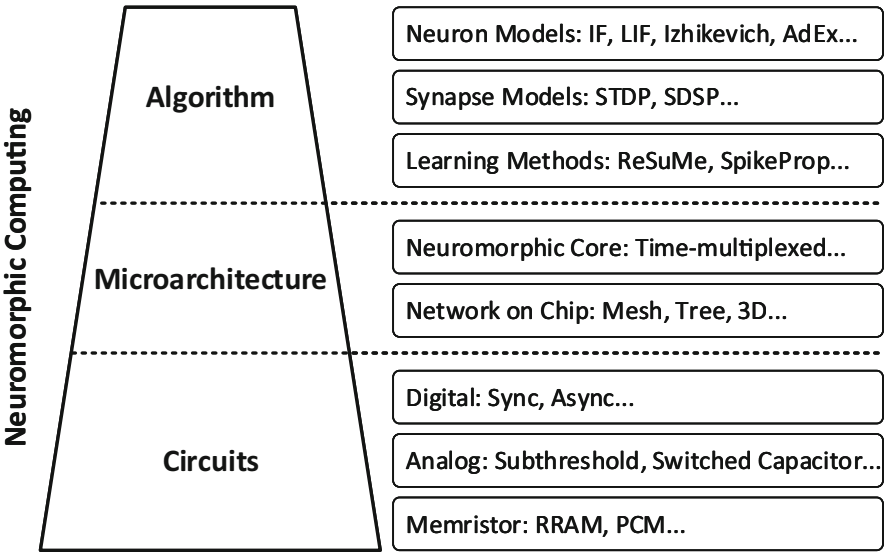


Fig. 2 Design abstractions of neuromorphic computing across algorithm, microarchitecture, and circuits

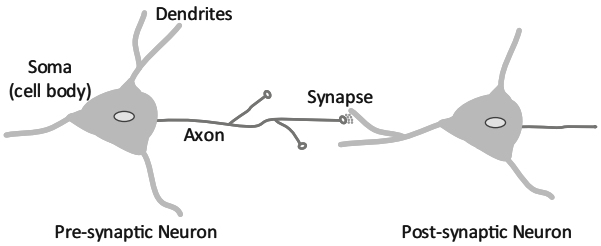


Fig. 3 Biological neuron and synapse

There are three main differences in computing principles between the brain and ANN accelerators. First, the ANN processes information in precise multi-bit values, while the brain processes spikes or events. Second, computation (i.e., the neurons) and storage (i.e., the synapses) are co-located in the brain. By contrast, the computation and storage in the ANN accelerator are separated. Third, the connectivity of neurons in the brain is three-dimensional, which is much more massive than that of the ANN accelerator.

Different neuron and synapse models have been proposed to describe the biological behaviors at different levels of biological plausibility, while the neuromorphic computing needs to mimic only the key parts which are essential for computation. A typical neuromorphic computing model is shown in Fig. 4.

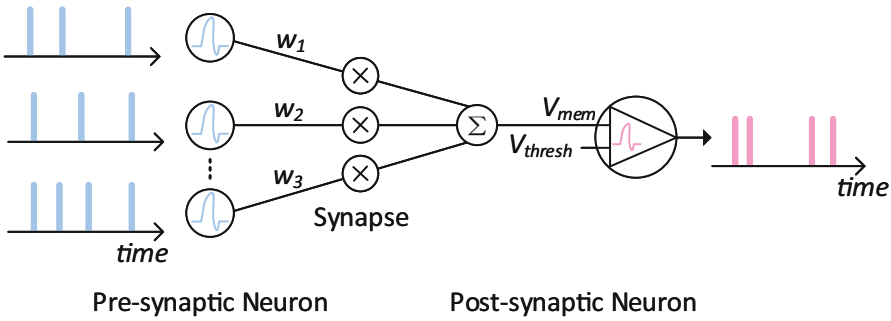


Fig. 4 Typical neuromorphic computing model

Neuron Models The widely used neuron models in neuromorphic computing accelerator are some hardware-friendly (with less implementation cost) computing models, such as leaky intergrate and fire (LIF), quadratic LIF (QIF), Izhikevich, and so on. These models can be described as in the form of ordinary differential equations. The membrane potential V_{mem} of the LIF model that is most commonly used can be calculated according to the following differential equation: Equation 1.

$$\begin{cases} \tau \frac{dV_{\text{mem}}}{dt} = \sum_i V_i w_i - V_{\text{mem}} + V_{\text{rest}}, \\ V_{\text{mem}} \quad = V_{\text{rest}}, \quad \texttt{if} \quad V_{\text{mem}} \geq V_{\text{thresh}} \end{cases} \tag{1}$$

where V_{rest}, V_{thresh}, and τ are the membrane's resting potential, threshold, and time-constant, respectively. w_i is the weight of the synapse. The dynamics of LIF-spiking neurons are shown in Fig. 5. V_i is the input synaptic voltage (the value is a binarization, where 1 and 0 represent a fired spike is received and no spike is received, respectively), and it will be summed until the neuron generates a spike when membrane potential across the threshold V_{thresh}. A refractory period is followed after spike generation, during which the membrane potential is kept at resting potential V_{rest}, and the input voltage V_i will not be added during this refractory period. This LIF model exhibits three neuro-computational properties, i.e., tonic spiking, class 1 excitable, and integrator.

The QIF model, which is also known as the theta-neuron, is higher of biological plausibility than LIF but also with higher hardware implementation cost. The QIF model can be presented as Equation 2. Compared to LIF, QIF model exhibits more neuro-computational properties like spike latency, threshold variability, and bistability of resting.

$$\begin{cases} \tau \frac{dV_{\text{mem}}}{dt} = \sum_i V_i w_i - \alpha(V_{\text{mem}} - V_{\text{rest}})(V_{\text{mem}} - V_{\text{thresh}}), \\ V_{\text{mem}} \quad = V_{\text{rest}}, \quad \texttt{if} \quad V_{\text{mem}} \geq V_{\text{thresh}} \end{cases} \tag{2}$$

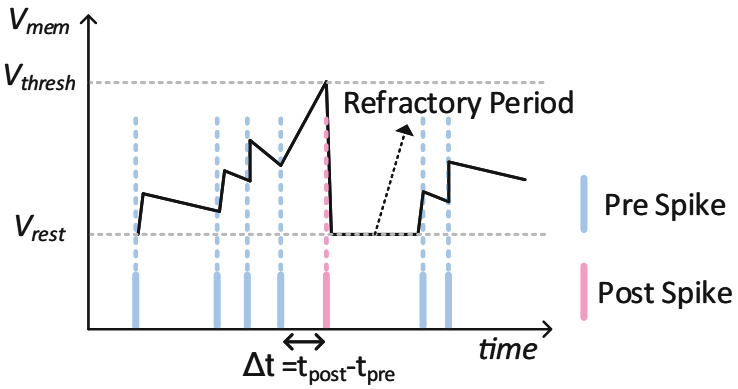


Fig. 5 The dynamics of LIF-spiking neurons

Izhikevich model is proposed in (2003), which can exhibit firing patterns of all known types of cortical neurons, and it can exhibit 20 neuro-computational properties. Izhikevich model is presented as Equation 3.

$$\begin{cases} \frac{dV_{\text{mem}}}{dt} = \sum_i V_i w_i + 0.04V_{\text{mem}}^2 + 5V_{\text{mem}} + 140mV - U, \\ \frac{dU}{dt} = a(bV_{\text{mem}} - U), \\ V_{\text{mem}} = V_{\text{rest}}, \quad \texttt{if} \quad V_{\text{mem}} \geq 30mV \\ U = U + d, \quad \texttt{if} \quad V_{\text{mem}} \geq 30mV \end{cases} \tag{3}$$

where mV stands for millivolt and U represents a membrane recovery variable that provides negative feedback to membrane potential V_{mem}. With different values of parameters a, b, V_{rest}, and d, this model exhibits different firing patterns. When the membrane potential V_{mem} reaches $30mV$, V_{mem} and U are reset to V_{rest} and $U + d$, respectively.

Synapse Models The fired spikes in presynaptic neuron transit to the postsynaptic neuron through synapses. In a neuromorphic computing accelerator, if the system does not attempt to explicitly model synapse biological behavior such as plasticity mechanism, the synapses are considered as scalar values or the weight that are stored in the memory. For these accelerators without the plasticity mechanism, they are not able to do training or learning. To mimic the learning mechanism of the brain, the synapse model has to model the synapse biological behavior such as the plasticity mechanism. This plasticity mechanism will modify the neuron's strength or the weight value over time. One of the most used synapse models with plasticity mechanism is the spike-timing-dependent plasticity (STDP), and this biological

behavior has been observed around the year 1998 (Bi and Poo 1998). Specifically, the weight of the synapse increases or be strengthened ("potentiated" in the field of neuroscience technology) if the spike fired by the presynaptic neuron arrives within a certain time window before the spike generation of the postsynaptic neuron. On the contrary, the weight of the synapse decreases or be weakened ("depressed" in the field of neuroscience technology) if the spike fired by the presynaptic neuron arrives within a certain time window after the spike generation of the postsynaptic neuron. STDP is also known as the unsupervised learning method in neuromorphic computing, and more details about STDP will be described in the next paragraph related to the learning method.

Learning Methods Currently, the effective learning or training methods for SNN are spike-based supervised, unsupervised, and semi-supervised methods. For the supervised learning methods, the data fed to SNN for training should be labeled. ReSuMe, tempotron, and backpropagation are the typical spike-based supervised learning methods. While inspired by the brain, the STDP-based algorithm is one of the most used unsupervised learning methods. Semi-supervised methods, such as spike-driven synaptic plasticity (SDSP), are less complex of implementation than STDP. Supervised learning methods such as ReSuMe and tempotron can train SNN with a single layer. But for complex neuromorphic computing like multilayer SNN, supervised learning methods are not effective learning methods. Spike-based backpropagation algorithm enables supervised learning for multilayer SNN, and it has been widely used for feedforward SNNs. Similar to the backpropagation in feedforward ANNs, the spike-based backpropagation algorithm (as shown in Fig. 6) backpropagates the computed gradient for the weights of the network for a single input-output example (here is spike train). However, unlike the ANN, the fired spikes in SNN are non-derivable due to their discontinuity. To overcome this difficulty, most works, such as SpikeProp (Bohte et al. 2002), Multi-SpikeProp (Ghosh-Dastidar and Adeli 2009), and NormAD (Anwani and Rajendran 2015), estimate a differentiable approximate function for the spikes, and thus, the gradient descent can be performed.

The bio-inspired STDP-based learning method is a temporally asymmetric form of the Hebbian learning rule, where the weight of synapses is updated

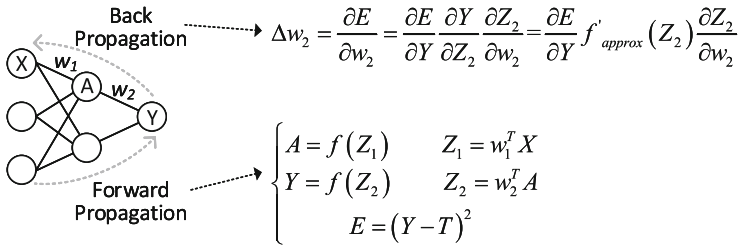


Fig. 6 Spike-based backpropagation supervised learning method

according to the relative timing of a particular neuron's received and transmitted spikes. As shown in Fig. 7, if the spike fired by the presynaptic neuron occurs immediately before the postsynaptic neuron spike generation, then the synapse connecting this presynaptic neuron and the postsynaptic neuron is strengthened. In other words, if the spike generated by the presynaptic neuron persistently takes part in firing the postsynaptic neuron, then the weight of this synapse increases. While if the presynaptic neuron's spike persistently independent of firing its postsynaptic neurons, then the weight of this synapse decreases. The formulation of this STDP-based learning method is presented in Equation 4, where α_+/α_- and τ_+/τ_- are learning rates and time-constants governing the weight change Δw, and Δt is the time difference between presynaptic and postsynaptic neuron spikes, i.e., $\Delta t = t_{\text{post}} - t_{\text{pre}}$.

$$\Delta w = \begin{cases} \alpha_+ \exp(\frac{-\Delta t}{\tau_+}), & \texttt{if} \quad \Delta t > 0 \\ -\alpha_- \exp(\frac{\Delta t}{\tau_-}), & \texttt{if} \quad \Delta t < 0 \end{cases} \tag{4}$$

Other STDP variants with model reduction such as SDSP have also been studied. The illustration of the SDSP-based learning method is shown in Fig. 8 and the synaptic weight updates following the equation of Equation 5. Here, the a and b are jump sizes, θ_m is the voltage threshold, and θ_1, θ_2, and θ_3 are the thresholds on the calcium variable. Different from the STDP learning method that relies on the relative presynaptic neuron and postsynaptic neuron spike times, the SDSP learning method updates the weight each time when a presynaptic neuron spike occurs. The synaptic weight increases if the membrane potential at the time of presynaptic neuron spike

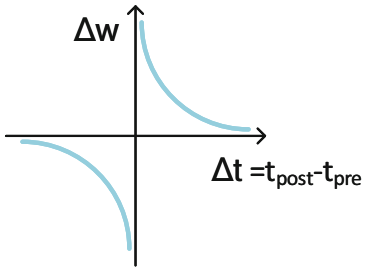


Fig. 7 STDP-based learning method

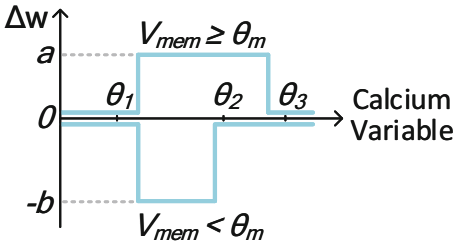


Fig. 8 SDSP-based learning method

$V_{\text{mem}}(t_{\text{pre}})$ is larger than the threshold θ_m, and if the calcium concentration $Ca(t_{\text{pre}})$ at the time of presynaptic neuron spike is between θ_1 and θ_3, while the synaptic weight decreases by b at the time of presynaptic neuron spike if Vmem$V_{\text{mem}}(t_{\text{pre}})$ is smaller than the threshold θ_m, and $Ca(t_{\text{pre}})$ is between θ_1 and θ_2.

$$\Delta w = \begin{cases} +a, & \texttt{if} \quad V_{\text{mem}}(t_{\text{pre}}) \geq \theta_m \quad \texttt{and} \quad \theta_1 \leq Ca(t_{\text{pre}}) < \theta_3 \\ -b, & \texttt{if} \quad V_{\text{mem}}(t_{\text{pre}}) < \theta_m \quad \texttt{and} \quad \theta_1 \leq Ca(t_{\text{pre}}) < \theta_2 \end{cases} \tag{5}$$

Microarchitecture for Neuromorphic Computing

Framework Inspired by neuroscience, the state-of-the-art architectural framework of neuromorphic chip or SNN accelerator, as shown in Fig. 9, is constructed by tiling neuromorphic cores or neurosynaptic cores. Since the essential computing units (i.e., the neuron) and storage elements are collocated in each neuromorphic core, this kind of architecture mitigates the von Neumann bottleneck, and it is easy to scale up to a highly parallel architecture and efficiently implement scale-out neural networks.

Neuromorphic Core The state-of-the-art neuromorphic core mainly consists of a crossbar array, neurons, storage elements, and some peripheral circuitry, which is shown in Fig. 10. The neuron circuitry mimics the behavior of the biological

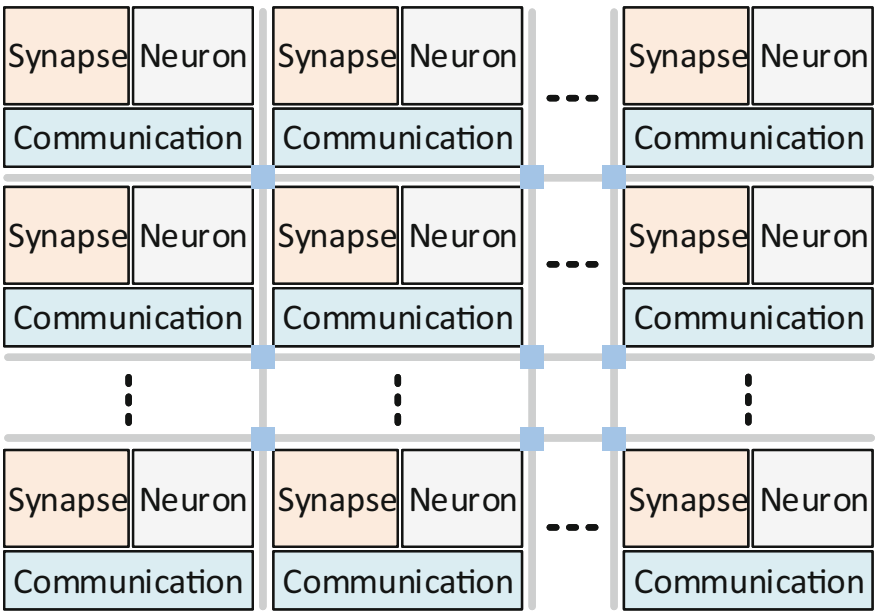


Fig. 9 The architectural framework of neuromorphic chip or SNN accelerator

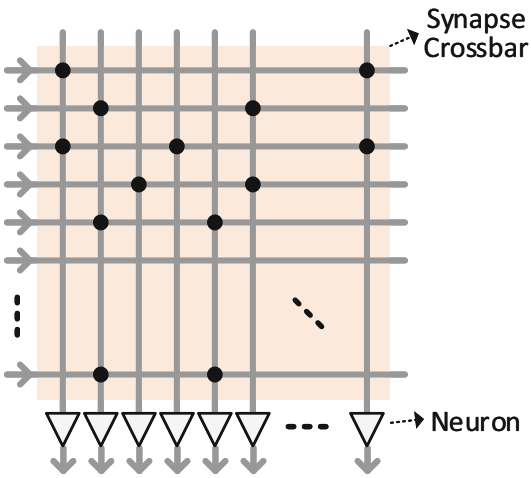


Fig. 10 The architecture of neuromorphic core

neuron, such as LIF, QIF, and Izhikevich. The crossbar array mimics the dense local connectivity of neurons, and each cross-point in the crossbar array is the synapse memory. Combined with the learning block, it mimics the biological synapse behaviors like STDP and SDSP. The local storage elements in each neuromorphic core are used to store the synaptic weight, lookup tables for routing information, and local data. The peripheral circuitry implements the communication interface and also the drive circuits that control the input wires (i.e., axons) and output wire (i.e., dendrites).

NoC Although the local connectivity in a typical neuromorphic core is bio-inspired dense connectivity, the communication between neuromorphic cores is also significant to scale up the neuromorphic chip. Network-on-chip (NoC) with routers is widely used in the neuromorphic chip for communication. The state-of-the-art topology of these NoCs mainly are two-dimensional (2D) connectivity, such as mesh topology, tree topology, and tree-mesh hybrid topology, and some neuromorphic chips even use brain-like three-dimensional (3D) connectivity. However, for the neuromorphic chip built on a single chip, its NoC is still limited to 2D connectivity because connectivity in a typical silicon fabrication process is two-dimensional.

The 2D-mesh topology where the routers are connected to neighboring routers is one of the most widely used routing schemes. Figure 11a shows the typical 2D-mesh or grid routing scheme adopted by TrueNorth and Loihi, where each router connects to other routers in four cardinal directions. 2D mesh with triangular facets, as shown in Fig. 11b, is also used to connect neuromorphic cores like SpiNNaker neuromorphic chip. 2D-mesh routing is easy to avoid deadlock by using a dimension-order routing algorithm, while an arbiter has to be built in the router to perform arbitration when the router receives multiple communication packets at the same time. Besides, to prevent any communication packets from being dropped, backpressure scheme has to be designed. For example, if the router is waiting to

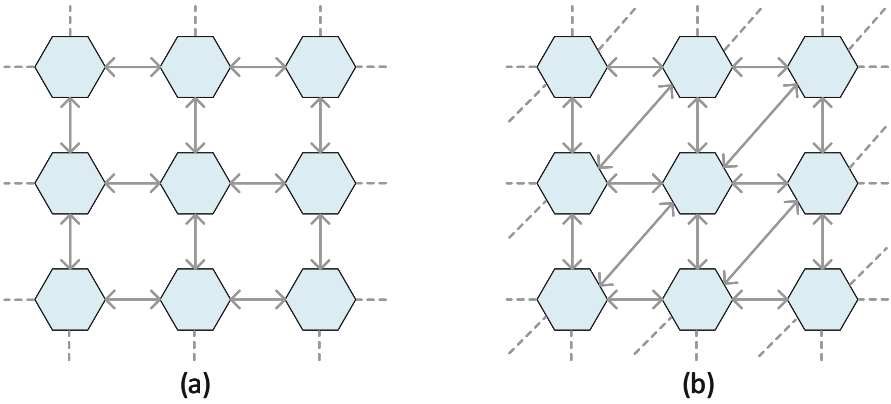


Fig. 11 2D-mesh routing scheme (**a**) with four cardinal, (**b**) with triangular facets

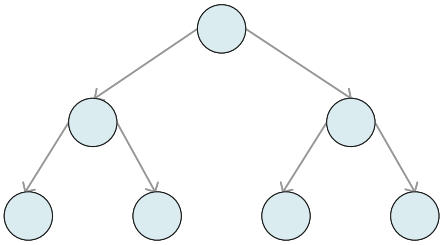


Fig. 12 2D-tree routing scheme

transmit outgoing communication packets, the backpressure scheme should be able to prevent this router from receiving new packets.

2D-tree topology is another 2D routing scheme that has been used in neuromorphic chips such as Neurogrid. Figure 12 shows a hierarchical tree topology, where up-down routing is adopted to avoid deadlock. In 2D-tree topology, with an up-down routing structure, the communication packets are transmitted to the root node, and then these communication packets can be duplicated child nodes.

2D-mesh and 2D-tree routing scheme have their advantages and disadvantages. 2D-mesh routing scheme has larger channel bisection and more connection paths, leading to higher throughput of communication packets. However, a 2D-mesh routing scheme normally has a longer latency since it requires more hops to transmit a communication packet to its destination neuromorphic core, especially for a neuromorphic chip with a large number of neuromorphic cores. 2D-tree routing scheme needs fewer hops to transmit a communication packet to its destination neuromorphic core, resulting in shorter latency, while the throughput of the 2D tree is lower because there are fewer connection paths.

A hybrid routing scheme has been designed to combine the benefits of 2D mesh with high throughput and 2D tree with low latency. As shown in Fig. 13, the top

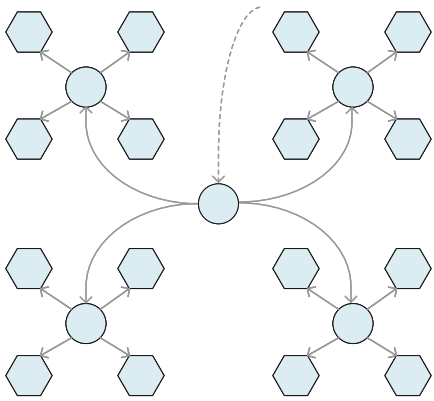


Fig. 13 Hybrid routing scheme

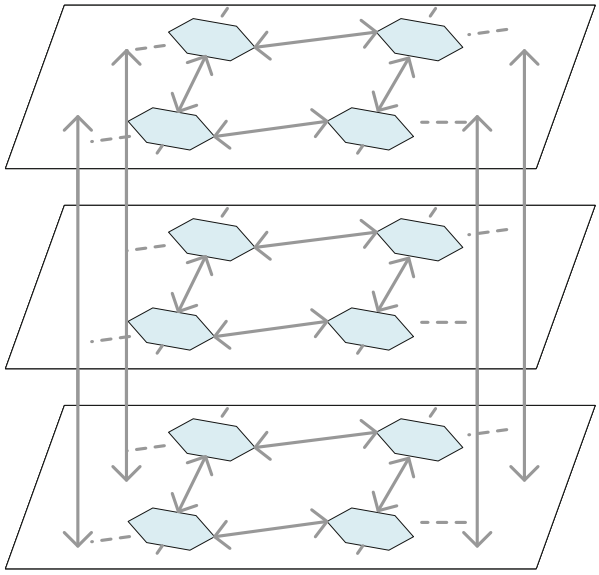


Fig. 14 3D NoC routing scheme

level of routers is connected with 2D-tree topology, while the low level of routers is organized as 2D-mesh topology. The DYNAPs neuromorphic chip adopts this hybrid routing scheme.

When compared to the network structure of the brain, the 2D NoC is still limited to connectivity and throughput. 3D NoC which mimics the 3D structure in the brain takes a step closer toward bio-plausible implementation. A 3D NoC, as shown in Fig. 14, includes the vertical crossbar and the horizontal 2D NoC. This 3D NoC improves the performance of the neuromorphic chip by shorter latency and higher throughput. However, the state of the art of the 3D NoC still cannot be implemented

in a typical single silicon chip since the chip fabrication process is two-dimensional. Take 3D NoC in LaCSNN neuromorphic chip, for example; it is built by vertically stacking multiple FPGA boards through a high-speed Terasic connector (Yang et al. 2018).

AER Protocol Since the spiking rate of a biological neuron is much lower than that of electronic circuits in orders of magnitude, and the biological axonal delays and neuron time constants are larger than electronic circuits' propagation and transition delay, the NoC in the bio-inspired neuromorphic chip or SNN accelerator normally uses address event representation (AER) protocol to transmit communication packets from one neuromorphic core to another. Each neuron has a unique address to enable communication base on the AER protocol. When a neuron generates a spike or an event, this spike information including the spiking neuron address and the time of spiking will be transmitted to the destination neurons.

A simplified AER-based communication system is shown in Fig. 15. Whenever the neuron on the transmitter side generates a spike or an event, the spike information including the corresponding neuron address that will be encoded and sent over the data bus to the destination receiving neurons. The decoder of the receiving neurons decodes the incoming AER packets and reconstructs the sequence of the spikes. Therefore, in this basic AER-based communication system, the spike is explicitly encoded by its address, while the time of spiking is implicitly encoded by the time when its address is sent to the data bus.

When compared to the frame-driven approach of the ANN accelerator, this AER-based NoC communication system of the SNN accelerator is easier to scale. It is possible to connect any number of neuromorphic cores as long as the routers can manage the communication packet of the AER protocol. Thus, AER-based NoC is suitable for not only the on-chip communication but also the chip-to-chip communication, enabling larger-scale design.

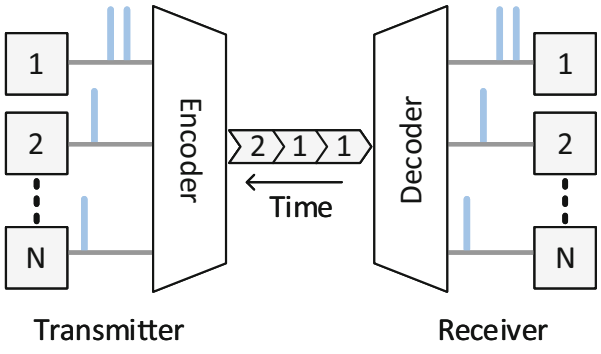


Fig. 15 Simplified AER-based communication system

Circuit-Level Design Considerations

The main building blocks for the neuromorphic chip or SNN accelerator are neuromorphic core and AER-based NoC. When implementing these building blocks, the power-performance-area (PPA) is the main design consideration. It is significant to choose a design technique that will not only mimic the behavior of the biological brain but also with a good PPA (i.e., low power, high performance, and small area). The state-of-the-art neurons are implemented using analog circuits or digital circuits. Besides, to store synaptic weight and some local data, local memory like SRAM has been employed in the neuromorphic cores, enabling near-memory computing. Moreover, neurons and synapses can be implemented using memristive technologies, which implement computing and storage in one memristive device, while the AER NoC is normally implemented using digital circuits but can be either synchronous or asynchronous circuit design technique.

Neurons Using Analog Circuits Various analog circuit designs could be used to build a neuromorphic core including neuron and synapse, such as subthreshold circuits, above-threshold circuits, switched-capacitor circuits, and so on. Metal oxide semiconductor (MOS) transistors that operate in subthreshold or weak-inversion regimes can be used to implement a first-order low-pass filter (LPF) and thus can faithfully model bio-plausible temporal dynamics, since the current-voltage characteristics of these transistors are exponential. For example, Fig. 16 shows the circuit schematic of the adaptive exponential IF neuron (Chicca et al. 2014). It consists of an input differential pair integrator as the LPF, a spike generating amplifier, a spike reset circuit with refractory period functionality, and a spike-frequency adaptation mechanism. MOS transistors $M_{L1} - M_{L3}$ model the leak conductance of the neuron, and they produce exponential subthreshold

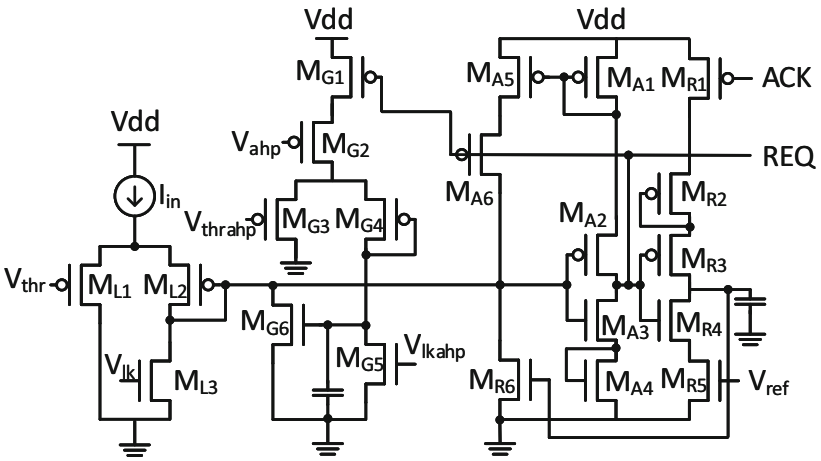


Fig. 16 The adaptive exponential IF neuron implemented by subthreshold circuits

dynamics in response to constant input currents. The capacitor C_{mem} represents the neuron's membrane capacitance, and the activation and inactivation dynamics of the sodium channel are modeled by the positive-feedback circuits, i.e., MOS transistors $M_{A1} - M_{A6}$. The potassium conductance and refractory period functionality are modeled by the $M_{R1} - M_{R6}$, while the spike-frequency adaptation mechanism is implemented by the $M_{G1} - M_{G6}$. These MOS transistors model the neuron's calcium conductance which generates the after-hyperpolarizing current that is proportional to the spike's firing rate. There are many biases, such as the V_{thr}, V_{lk}, V_{thrahp}, V_{ahp}, V_{lkahp}, and V_{ref}, in this neuron circuit schematic that can be tuned. By changing these biases that control the neuron's time constants, refractory period, and spike-frequency adaptation dynamics, this circuit can model different spiking behaviors ranging from regular spiking to bursting.

Although the subthreshold circuits can be used to build biophysically realistic models with ultralow power consumption (ranging from between fractions of pico- to hundreds of nano-amperes) and realistic time constants, they suffer from high device mismatch especially for the large-scale neuromorphic chip or SNN accelerator. The neuromorphic core implemented by the above-threshold circuits and switched-capacitor circuits are more robust but with higher power consumption than that of subthreshold circuits. Figure 17 shows an example of the Izhikevich neuron implemented by the above-threshold circuits (Wijekoon and Dudek 2008). The voltages across capacitors C_v and C_u represent the membrane potential and slow variable of the Izhikevich model, respectively. The MOS transistors $M_1 - M_5$ together with the membrane capacitor C_v construct the membrane potential circuits. The input currents, the positive feedback current of M_3 generated by spike, and the M_4 leakage modeling the recovery variable of Izhikevich neuron are integrated on the membrane capacitor. The positive feedback current is generated by M_1 which is controlled approximately quadratically by the membrane potential and mirrored by the current mirror ($M_2 - M_3$). When a spike is fired, the analog comparator ($M_9 - M_{14}$) will generate a reset pulse on the gate of transistor M_5, and then the membrane potential will be hyperpolarized to a voltage value determined by the voltage at node c. Transistor M_1, M_2, M_6, M_7, and M_8 built the slow variable circuit, where the current of M_7 is determined by the membrane potential and M_6

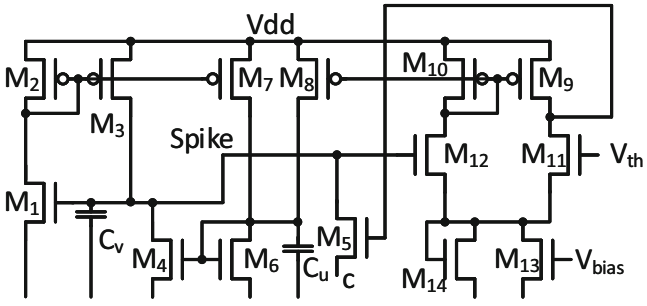


Fig. 17 The Izhikevich neuron implemented by above-threshold circuits

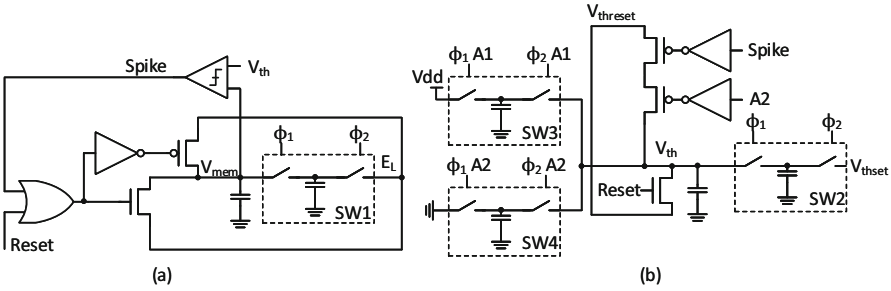


Fig. 18 The Mihalas-Niebur neuron implemented by switched capacitor circuits (**a**) membrane potential circuit, (**b**) adaptive threshold circuit

provides a nonlinear leakage current. The size of these transistors and the value of the capacitors can be scaled to make the slow variable changes more slowly than the membrane potential.

Switched-capacitor circuits have been widely used in analog circuits to realize the variable resistor with a wide range of several orders of magnitude. This circuit technique can also be used to build the neuron model. Figure 18 shows an example of a switched-capacitor-based Mihalas-Niebur neuron that is a generalized version of the LIF model with an adaptive threshold (Folowosele et al. 2009). It consists of the membrane potential circuit (Fig. 18a) and the adaptive threshold circuit (Fig. 18b). An analog comparator is used to compare the membrane potential V_{mem} and the adaptive threshold V_{th}, and it will generate a spike when a membrane potential larger than the threshold. When a spike is generated, the membrane potential is reset to the resting potential determined by the voltage at node EL. The switched-capacitor $SW1$ switched by two nonoverlapping clocks $\phi 1$ and $\phi 2$ is used to model the conductance. The voltage divided by the two variable resistors realized by two switched-capacitor circuits in the adaptive threshold circuit block, i.e., $SW3$ and $SW4$, is used to generate the adaptive threshold.

Besides the neuron, the analog circuits can also be used to model the biological synapse. Figure 19 shows an example of STDP synapse using analog circuits (Indiveri et al. 2006), which can emulate the weight update curves in Fig. 7. The charge stored on the weight capacitor C_w represents the weight of each synapse, and the strength of the weight is inversely proportional to the voltage V_w at the capacitor. When the presynaptic neuron fires, i.e., the signal Pre on the gate of M_10 is a pulse, the voltage V_{rLTP} on the diode-connected transistor M_9 is copied to V_2, the gate voltage of M_{13}. The leakage current of M_{11} decays V_2 with time from its peak voltage. When the postsynaptic neuron fires, a spike or pulse turns on M_{12}. Thus, the weight is potentiated (i.e., V_w decrease) by an amount that reflects the time elapsed since the last presynaptic neuron spikes. Similarly, the weight is weakened if the circuit detects a noncausal interaction between a presynaptic neuron and a postsynaptic neuron-generated spike. When a postsynaptic neuron fires, a pulse on the gate of M_{15} charges the voltage V_1. The charge on the capacitor C_{dep} leaks

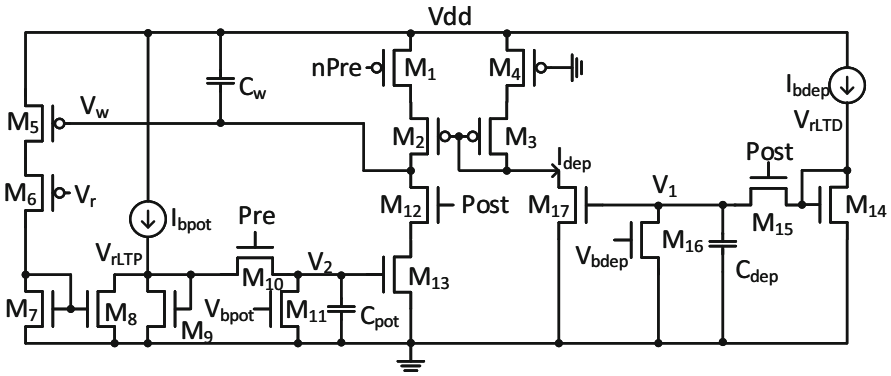


Fig. 19 The STDP synapse model implemented by analog circuits

through M_{16}. Then, a nonlinear current I_{dep} is sent to decay the voltage V_w. When the presynaptic neuron fired spike occurs soon enough after the postsynaptic neuron spike, V_w is increased to supply voltage, i.e., the weight strength decreases.

The circuits described above, including subthreshold circuits, above-threshold circuits, and switched-capacitor circuits show their ability to model different neurons and synapses in the neuromorphic core. Analog circuits especially the subthreshold circuits are suitable for modeling complex or biophysically realistic models since the transistors working in the subthreshold regime have exponential behaviors. However, compared to digital circuits, analog circuits tend to accumulate errors easily and are much more prone to process-induced variations in chip fabrication.

Neurons Using Digital Circuits Although the digital circuits, such as adders, multipliers, counters, and SRAM cells, can model complex neuron and synapse models, the computation tasks are heavy leading to a high cost of implementation. The state-of-the-art neuromorphic core using digital circuits tends to model simple biological dynamics like LIF. Both synchronous and asynchronous circuits have been used to build the neuromorphic core. Thanks to the mature standard ASIC design flow, most of the digital neuromorphic neurons are implemented using synchronous circuits which operate based on the clock. A typical LIF digital neuron using synchronous circuits is shown in Fig. 20. The integrator unit accumulates the synaptic weight when a spike occurs, while a leak is reduced from the accumulated membrane potential at every time tick. If the membrane potential value is equal to or larger than the threshold value, a spike is generated and transmitted, and the membrane potential is reset at the same time. The neuron operation proceeds until every spike from the previous timesteps is processed. All of these operations are updated at the rising or falling edge of the clock, i.e., the circuits operate synchronously.

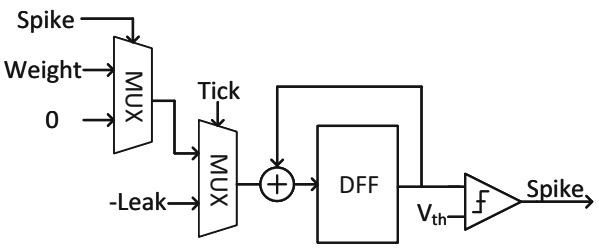


Fig. 20 Neuromorphic core using digital synchronous circuits

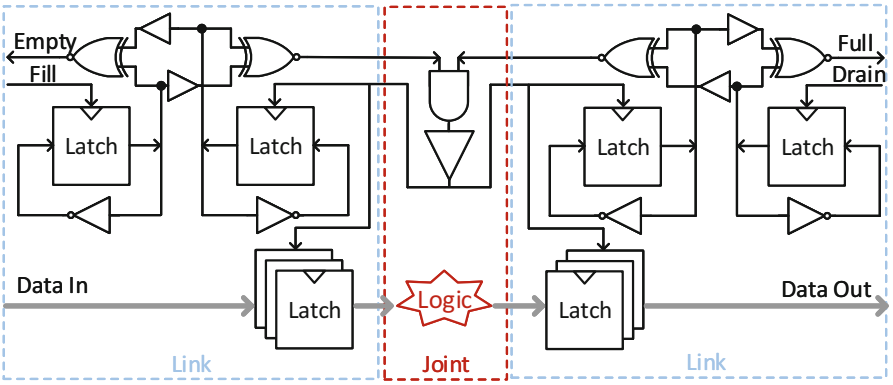


Fig. 21 (**a**) Click-based link-joint asynchronous circuit, (**b**) Neuromorphic core using a digital asynchronous circuit

Link-joint-based asynchronous bundled data circuits have been used to build neurons, as shown in Fig. 21 (Zhang et al. 2019). The links have a self-timed asynchronous controller, where handshake signals are used to realize transmitting and receiving data, while the joints perform logic computing such as accumulation, multiplexing, and comparing. The joint computes the data taken from its full input links and passes the computing results to some or all of its empty output links. As it is a spike-driven manner, the neuromorphic cores achieve good energy efficiency.

Neuromorphic Core Using Memristive Devices In memristive devices, the information is stored in the form of resistance or conductance. Their resistance values are altered when applying an appropriate electrical pulse to the devices. Besides,

if successive programming pulses of the same amplitude apply to memristive devices, their resistance values will be incrementally increased or decreased. These characteristics make the memristive devices suited for building the neuromorphic core. Various types of memristive devices, such as phase-change memory (PCM) (Tuma et al. 2016), metal-oxide-based resistive RAM (RRAM) (Waser et al. 2009), and spin-transfer-torque magnetic RAM (STTMRAM) (Hosomi et al. 2005), have been used to build neuromorphic core.

These memristive devices, organized in a crossbar architecture (as shown in Fig. 22), can mimic two important behaviors of the biologic synapse, i.e., the synaptic efficacy and synaptic plasticity. Synaptic efficacy refers to the generation of a synaptic output based on incoming spikes. This means that incoming spikes (in the form of voltage pulse) from all presynaptic neurons are multiplied by the stored weights (in the form of resistance) and the resulted currents that flow through the devices are summed. Synaptic plasticity means that the weights of the synapses are modulated according to a particular learning rule like STDP. Other neuronal behaviors such as stochastic neuronal dynamics also can be implemented in the memristive devices. For example, the PCM-based neurons exhibit significant interneuronal as well as intraneuronal randomness, which enables them to mimic the stochastic neuronal dynamics at the device level. Besides, such memristive device-based crossbars can be connected to build large-scale neuromorphic chips or SNN accelerators using spike-driven NoC, and these processors will be with features of in situ in-memory computing. Although state-of-the-art emerging memristive devices are well suited to implement neuronal dynamics, several challenges need to be overcome to enable mass production. These challenges such as RRAMs are prone to device-to-device and cycle-to-cycle variations; PCMs suffer from high write-current and resistance drift over time; and STTMRAMs suffer from a substantially lower dynamic range in programmable conductance states.

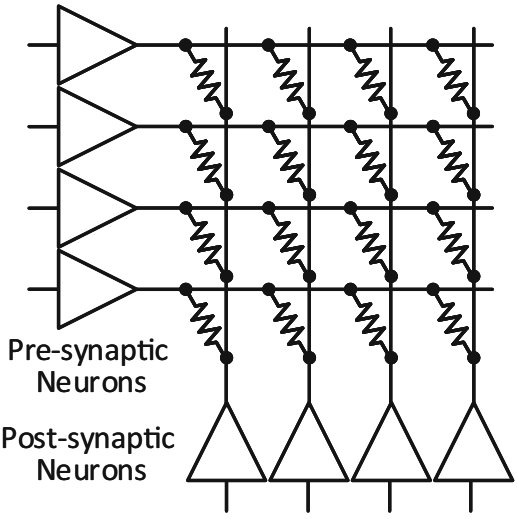


Fig. 22 Neuromorphic core using memristive devices

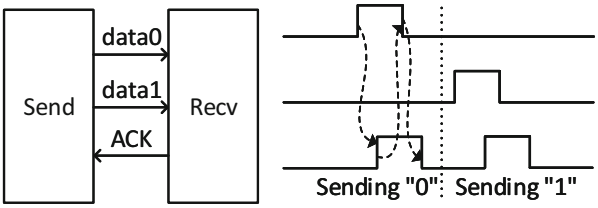


Fig. 23 Asynchronous four-phase handshake protocol

Communication Using Asynchronous Circuit Spike-driven nature is one of the main features of neuromorphic computing, where the computations are performed only when a spike occurs and thus make the neuromorphic chip to be energy efficient. Unlike synchronous circuits where all registers storing data are updated simultaneously upon arrival of the global clock edge, asynchronous circuits operate by using a handshaking mechanism without a clock to transmit or receive data. Besides, routing a synchronous clock across a very large chip is very challenging and with large power and chip area overhead. Hence, asynchronous circuits are more suitable for building communications in the neuromorphic chip. Quasi-delay-insensitive (QDI) circuits (Martin 1990) are widely used in neuromorphic chips to realize asynchronous communication. The QDI design style, which bases on the synthesis procedure proposed by Martin (Wong and Martin 2003), decomposes communicating hardware processes (CHP) into many fine-grained circuit elements that operate in parallel. Normally, a four-phase handshake protocol is used to synchronize the communicating tokens over delay-insensitive channels. Figure 23 shows a single bit of such communication protocol. When the transmitter transmits a "1" or "0" value, the corresponding data line (data0 and data1) is asserted. Subsequently, the receiver will send acknowledge signal after it receives the data. This acknowledgment will reset the data line, and finally, all the signals are reset.

Asynchronous Design Methodology Since the majority of the conventional chips are designed only using synchronous circuits, the design methodologies or EDA tools are optimized for synchronous design style. It is crucial to develop an asynchronous design methodology, especially for large-scale asynchronous chips like neuromorphic chip. Here, the design methodology developed for Intel's Loihi (Davies et al. 2018) will be described. In this asynchronous design methodology, designs are specified in the CAST (Martin and Nyström 2004) and Communicating Sequential Processes (CSP) languages (Lines et al. 2018) and are followed a top-down decomposition process. By using the CAST asynchronous synthesis tools, modules described in CSP are automatically converted into Verilog with the addition of circuits that enable asynchronous implementation. The topology of each module implementation is derived from CSP, and each module contains one stage of computing logic, bundled data pipeline stage, state variables loop back through two pipeline stages such as the handshake signals, as demonstrated in Fig. 24. Such

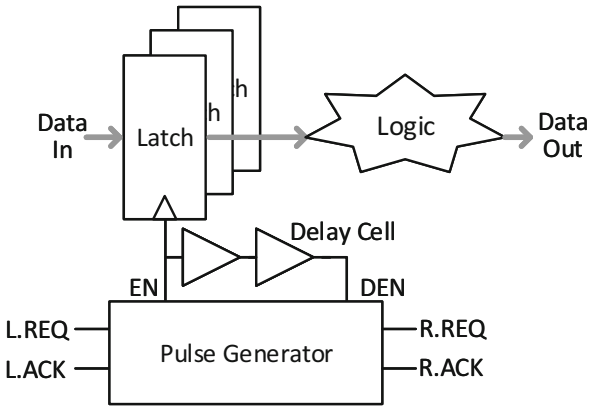


Fig. 24 Asynchronous design modules

Verilog representations converted from CSP language are compatible with standard EDA tools. These converted Verilog can be synthesized utilizing a standard-cell library with the addition of some special customized cells, such as C-elements with combinational keepers, and tuneable delay line cells. To facilitate convergent timing closure, all timing constraints apply only to neighboring, physically proximate pipeline stages at each level of layout hierarchy. Hence, there is no unique clock distribution layout is required, and timing analysis is not needed for different neuromorphic cores.

The logic simulation for asynchronous circuits is also important during chip design. Production rule simulator (PRSIM) (Akopyan et al. 2015) has been developed for asynchronous circuits' logic simulation. The input to PRSIM is a netlist based on production rules, which can be viewed as a sequence of spikes in neuromorphic computing. All spikes are stored in a queue, and a timestamp is attached to the spike when its preconditions become true. If the timestamp of a spike coincides with PRSIM's running clock, the production rule (i.e., the spike) is executed to verify correct circuit behavior.

Summary In summary, the neuromorphic circuits including the neurons, synapse, and AER-based NoC can be realized by various circuit techniques, such as analog circuits (above-threshold, subthreshold, and switched-capacitor), memristive devices, and digital circuits (both synchronous and asynchronous). When choosing the type of circuit technique to implement neuromorphic chips, several circuit-level design facts should be taken into consideration, such as the power efficiency, performance, chip area, stability, the ability of bio-plausible, available EDA tools, manufacturing maturity, and so forth.

Prominent Neuromorphic Chips

Since Carver Mead initiated the field of neuromorphic engineering (Mead 1990), there are many breakthroughs in neuromorphic computing. In this section, some prominent neuromorphic chips or SNN accelerators will be described. In terms of their main applications, the state-of-the-art neuromorphic chips can be divided into two categories, i.e., to assist computational neuroscience in emulating brain activity and to accelerate SNNs performing commercially important tasks like classification and recognition. There are many large-scale neuromorphic chips with complex neuronal mechanisms that have been built for simulation of the biological brain, such as SpiNNaker, Neurogrid, BrainScales, LaCSNN, and so on. While there are more and more neuromorphic chips, such as TrueNorth, Loihi, ODIN, Tianji, and so forth, have been designed based on simplified neural and synaptic primitives (e.g., LIF neuron model and STDP synapse model) to perform commercial tasks.

SpiNNaker

SpiNNaker (Furber et al. 2014; Liu et al. 2018) is a large-scale digital neuromorphic chip, which is a part of the Human Brain Project (HBP) and developed by the University of Manchester. Instead of building custom circuits modeling the neuron and synapse, SpiNNaker systems use processor core to model biologic behaviors, approaching the complexity of the brain in real-time. Hence, the processor core with some local memory can be used to simulate complex models of neurons (such as LIF and Izhikevich) and synapses (such as STDP). The SpiNNaker is based on massively parallel computation with up to one million processor cores. They communicate with each other through very small packets but with high fan-in and fan-out connectivity, which mimics the human brain. Besides, the SpiNNaker is an event-driven system, where a message or event arriving at a processor core will trigger an interrupt that queues the event for processing by that core. The system has been designed to efficiently process small packets, which means it can keep the event queue not much larger than one. Moreover, SpiNNaker provides a software stack that can program the spiking network implemented with the PyNN (PyNN is a Python library supporting the portability of network designs between various neuronal simulators and hardware). Although SpiNNaker is highly programmable, it is not energy efficient and not fast especially for simulating complex neurons and synapses. The first generation of SpiNNaker is constructed from ARM968 processor cores. Eighteen of these processor cores with 96 KB of local memory, a packet router, and some other support peripherals are fabricated on a die. This die and another die with a 128 MB low-power SDRAM as the shared memory are stacked onto one package substrate and interconnected with gold wire bonding within a chip. Forty-eight of such chips are assembled on one board, with up to 864 processor cores. The chip-to-chip communication on the board is a custom protocol employing

direct connections, wired in a hexagonal topology. The SpiNNaker system is built with many such boards, and they communicate through high-speed serial links. The largest machine of the first generation of SpiNNaker–comprising 1200 boards in ten machine room cabinets–incorporates over one million ARM968 processor cores, with a total power consumption of 75 kW, and it can do the simulation of 1% of the human brain. In the second generation of SpiNNaker, a more advanced ARM processor core (the ARM Cortex M4F) and a more modern process technology (22 nm) with dynamic voltage and frequency scaling technique have been used to improve the performance and energy efficiency. The number of the ARM processor core per chip is up to 144. Besides, it integrated MAC accelerator. The goal of the second generation of SpiNNaker is to do the simulation of the complete human brain.

Neurogrid

Neurogrid (Benjamin et al. 2014) is a neuromorphic system for simulating large-scale neural models in real time to assist computational neuroscience in emulating brain activity. It is designed based on analog/digital mixed-signal circuits and developed by Stanford University. Neurogrid can emulate the biological soma, axonal arbors, dendritic trees, and synapses. Since the current-voltage characteristics of MOS transistors operating in the subthreshold regime are exponential, Neurogrid employs analog circuits that operate in such regime to implement the neuromorphic core and directly emulate various neuronal and synaptic dynamics. This implementation with analog circuits can also make Neurogrid very energy-efficient. The analog circuits emulating soma are dedicated, but circuits emulating axonal arbors, dendritic trees, and synapses are shared in the same neuromorphic core. With the shared dendritic trees for local connections, each shared synapse is connected to neighboring neurons, mimicking the structure of overlapping dendritic trees of the biological neural networks. Hence, Neurogrid is suitable for neural networks where most of the neurons' connections are local connections, such as the neocortex. While Neurogrid uses digital circuits to realize its AER-based communication system between neuromorphic cores, including the transmitter, receiver, router, and local RAM. All these digital circuits are asynchronous that emulate the brain's characteristic of event-driven, i.e., the neuromorphic cores are active only when a spike occurs. The neuromorphic cores are connected to a tree routing structure. The software stack of Neurogrid performs interactive visualization, composed of a user interface, a hardware abstraction layer, and driver components. Each neuromorphic core of Neurogrid consists of 256×256 neurons and is fabricated on a 12×14 mm2 die/chip in a 180 nm CMOS process technology. The full Neurogrid board comprises 16 such neuromorphic core chips connected with a tree multicast routing structure, which can simulate one million neurons with billions of synaptic connections in real time. Its power consumption is roughly 3 W.

BrainScales

BrainScales system (Schemmel et al. 2010; Friedmann et al. 2016), partly supported by the HBP, targets the emulation of most of the neural systems modeled in contemporary computational neuroscience at accelerated time. Compared to the biological real time, the acceleration factor of BrainScales ranges from 103 to 105 for spiking neural network emulations. In other words, to emulate 10,000 s of biological behaviors, the BrainScales system only needs about 1 s. BrainScales is designed based on analog/digital mixed-signal circuits. A complex biological neuron model, i.e., the adaptive exponential IF model which can be parameterized to exhibit diverse firing patterns, is implemented by the analog circuits. The synapse and its plasticity function have been implemented by the analog/digital mixed-signal circuits. The event-driven communication system is realized by the asynchronous digital circuits. The smallest silicon block of the BrainScales system is the High-Input Count Analog Neural Network (HiCANN) chip, consisting of neuron block (up to 512 neurons), synapse array (up to 14,000 synapses), routers for interconnections and other necessary supporting circuits. Multiple of these HiCANN chips are wired directly on the silicon wafer without cutting it into discrete elements to build the BrainScales system. The software/hardware framework of BrainScales has been developed using PyNN, which allows users to map spiking neural networks on the hardware for emulation. In the first generation of BrainScales system, HiCANN chips are fabricated in 180 nm CMOS process technology, in 2010. To realize wafer-scale integration 352 HiCANN chips on a single wafer, containing 4×106 synapses and up to 180,000 neurons, are wired. To enlarge the scale of the BrainScales system, wafer-to-wafer communication is also supported through FPGAs and 1 or 10 Gbit Ethernet links. The second generation of BrainScales system is designed based on 65 nm CMOS process technology. By combining general-purpose processors with fully custom analog circuits (correlation sensor block) on the die, it enables flexibility in implementable synapse learning mechanisms while keeping high efficiency, which is the major difference between the first and second generation of BrainScales. The correlation sensor block measures the time interval between pre- and postsynaptic neuron spikes. While the embedded processors can perform any functions that compute updates to the synaptic weights from the information of the correlation sensor.

LaCSNN

LaCSNN (Yang et al. 2018) is a high biological realism neuromorphic chip system implemented on FPGA with synchronous digital circuits. LaCSNN can simulate large-scale conductance-based spiking neural networks in real time. At the cellular level, the ionic channels, which play significant roles in neuronal activities, are efficiently realized by using multiplier-less digital circuits (i.e., the lookup tables). This implementation is based on a set of piecewise linear approximation-based

biologically realistic neuron models, including the subthalamic nucleus (STN), an external segment of the globus pallidus (GPe), internal segment of the GPe (GPi), and thalamocortical (TC) neuron models. The piecewise linear approximation approach is used to approximate the nonlinear functions of the ionic current model to reduce the required on-chip memory for lookup tables. LaCSNN applies the pipeline technique for enhanced operation frequency when building the piecewise linear approximation-based neuron models. At the network level, the AER-based communication system is adopted, but the digital topology of the NoC architecture is the 3D scalable topology, including the horizontal 2D NoC and the vertical crossbar. The chip-to-chip (connection between FPGAs) communication of each layer in the LaCSNN system employs the multicasting 2D mesh NoC architecture to provide high bandwidth. While the high-speed Terasic connector (HSTC) that is equipped on the FPGA board is used to implement the vertical crossbar, each HSTC connector is responsible for transmitting data from five nucleus processors. The LaCSNN system is realized by six Altera Stratix III 340 FPGAs. It can simulate up to one million neurons operating at 100 MHz, and up to 60 million synapses with a firing rate of 50 Hz in real time. The total power consumption is about 10.578 W. Thanks to the reconfigurability and scalability of the LaCSNN system, it can be used to interact with the corticobasal ganglia-thalamocortical loop, the implementation of other brain regions, or even the realization of the full mammalian brain.

TrueNorth

TrueNorth (Akopyan et al. 2015) is a low-power neuromorphic chip from IBM and is designed to solve commercially significant tasks such as speech and visual object recognition. To minimize the chip's active power, TrueNorth is designed using event-driven architecture, where power-hungry global clock networks are not required. Asynchronous digital circuits are employed to implement the AER-based communication and control logic to make TrueNorth event-driven. Neuromorphic cores are connected through scalable 2D mesh NoC, and each neuromorphic core can be configured to transmit the fired spikes to any other neurons. This AER-based communication and scalable routing scheme allow the integration of multiple chips. While the computational circuits, i.e., LIF neurons, in the neuromorphic core are realized by synchronous circuits, their clocks are controlled by the asynchronous control logic. Hence, the digital neuron circuit can be time-multiplexed to emulate the operation of up to 256 LIF neurons, which helps in minimizing the neuromorphic core area. In TrueNorth, memory and computation are collocated in the neuromorphic core to minimize the power consumed by data movement. Each neuromorphic core contains 256×410 bits of local SRAM memory to store the synaptic connections (256 bits), neuron potential and parameters (124 bits), spike destination (26 bits), and spike delivery tick (4 bits). Users can use the TrueNorth native Corelet language, together with the Corelet programming environment and a

library of composable algorithms to develop applications based on the TrueNorth chip. The TrueNorth chip consists of 4096 neuromorphic cores, containing one million LIF neurons and 256 million synapses, and it is a 5.4-billion-transistors chip fabricated in a 28 nm low-leakage CMOS process technology. The power consumption of TrueNorth is only 63 mW when it performs tasks of detection and identification for 5 objects on a 30-frame-per-second three-color video. IBM also uses the TrueNorth chip to build synaptic supercomputers, such as the NS16e-4 that contains 64 TrueNorth chips.

Loihi

Loihi (Davies et al. 2018) is a digital neuromorphic chip with a brain-inspired learning capacity, developed by Intel. The hardware-friendly LIF neuron model is implemented with asynchronous digital circuits in Loihi, and it is time-multiplexed. This allows the events or spikes to proceed as slow or as fast as the computation. Each neuromorphic core contains 1024 LIF neurons and 2 Mb of SRAM to store the connectivity information, configuration, and dynamic state of all neurons within the neuromorphic core. Inspired by biological reinforcement learning, the synaptic memory is highly reconfigurable, such as the weight precision (1 to 9 bits), delays (up to 63 timesteps), and the synaptic scratch variables are reconfigurable. Moreover, the neuromorphic core in Loihi supports flexible synaptic plasticity with microcode-programmable rules, such as the STDP, reward-modulated rules, rate-based Hebbian rules, and rules that mix activities filtered on different timescales. Loihi contains 128 of such neuromorphic cores, and they are connected through AER-based 2D mesh routing infrastructure that is realized by the event-driven asynchronous digital circuits. Loihi further extends the AER-based asynchronous communication to four off-chip interfaces, which helps scale the on-chip 2D mesh routing into chip-to-chip 2D mesh routing. Besides the neuromorphic cores, Loihi also integrates three microcontroller-class x86 processors. These processors are mainly used to convert the data format of conventional computing to spike or event-based data format. A set of software tools called NxSDK have been developed for Loihi. It provides APIs, compilers, debugging tools for programming Loihi, runtime for the lower layers, and interfacing to different third-party frameworks. The Loihi chip consists of 128 neuromorphic cores, containing 128k LIF neurons and 128 million synapses, and it is fabricated in Intel's 14 nm FinFET process technology with a chip area of 60 mm2. Intel has released three types of Loihi systems, i.e., Kapoho Bay USB form factor with 2 Loihi chips, Nahuku with 32 Loihi chips, and Pohoiki Springs system with 768 Loihi chips. Many applications have been demonstrated on Loihi, such as online learning of gestures using event-based image sensors, odor recognition and learning, closed-loop control for robotics, simultaneous localization and mapping of the robotics, and so forth.

ODIN

ODIN (Frenkel et al. 2018) developed by Catholic University Louvain is a digital neuromorphic chip with high-density embedded online learning for low-power, adaptive, and low-cost edge computing applications. ODIN can be programmed to model first-order LIF dynamics as well as the phenomenological model of the 20 Izhikevich behaviors. A global controller is used to time-multiplex the neuron logic circuit, where the neurons are updated sequentially. Although the neurons are implemented using a synchronous digital circuit design flow, they are entirely event-driven, and thus, they are not required to update at each timestep. ODIN implements the SDSP learning rule, which updates synaptic weight depending on only the state of the postsynaptic neuron (i.e., the comparing result between membrane potential and Calcium concentration) at the time of presynaptic neuron spiking. Thus, the computation of the SDSP can be offloaded to neurons, resulting in low-cost hardware implementation. The neuromorphic core integrates two blocks of single-port SRAM memory to store states and parameters of neurons and synapses. Each ODIN chip contains only one neuromorphic core, but more ODIN chips can be connected through their standard on-chip AER interface. ODIN contains 256 neurons and 64k synapses, and it is fabricated in 28 nm fully depleted silicon-on-insulator (FDSOI) CMOS technology, with a chip area of only 0.086 mm2. When deploying a single-layer fully connected 10-neuron network with SDSP learning onto ODIN chip, it achieves 84.5% classification accuracy for the Modified National Institute of Standards and Technology database (known as MNIST) data set and consumes only 15 nJ/inference at 0.55 V.

Tianjic

Tianjic (Pei et al. 2019) developed by Tsinghua University provides a hybrid, synergistic platform for accelerating both bio-inspired SNN and computer science-based ANN, and it is built with synchronous digital circuits. The basic building block of Tianjic is a unified functional core (called FCore), consisting of synapse, axon, dendrite, and soma circuits. Synapse and dendrite are shared, while axon and soma can be independently reconfigured, and thus, the FCore can be configured to compute both the coding format of SNN's spikes and ANN's digital number. The dendrite, performing membrane potential integration in SNN or MACs in ANN, shares the same calculators (in SNN mode, the multiplier is skipped). Besides, a unified data format is designed for the routing packet, and therefore, the routing infrastructure can be shared. Since the axon (as the input) and soma (as the output) are fully independently reconfigurable, it is possible to build homogeneous or heterogeneous networks by appropriately connecting multiple FCores. Besides the hardware, Tianjic also develops dedicated software tool chains that support applications both in SNN and ANN mode. These software tool chains contain several levels like a unified abstraction for programming and an automatic compiler

for mapping. Tianjic chip consists of 156 FCores, containing approximately 40,000 LIF neurons and 10 million synapses. These FCores are connected through 2D mesh NoC. The chip is fabricated using 28 nm processing technology with a chip area of 3.8×3.8 mm2. An unmanned bicycle system assembled with just one Tianjic chip has shown that it can simultaneously accelerate versatile algorithms and models and, thus, simultaneously perform multiple tasks including real-time object detection, tracking, voice control, obstacle avoidance, and balance control.

Architectures for Artificial Neural Networks

Although neuromorphic computing with complex models that mimic the human brain is a promising solution to future artificial intelligence, their practical application remains limited. For example, the accuracy of neuromorphic computing is still lower than that of the **Artificial Neural Network** (ANN) when performing commercial tasks; most of the conventional computing or storing products are real-valued-based hardware, which is not well suited for the event-based neuromorphic computing; even the programming framework of neuromorphic computing is different from the conventional programming framework. In terms of algorithm, software framework, and dedicated hardware, real-valued-based ANNs with simple models are more mature than neuromorphic computing. There are many ANN models like convolutional neural network (CNN), recurrent neural network (RNN), and transformer neural network. Along with these ANN models, there are also mature training algorithms. For example, backpropagation is a widely used training method. These ANNs can be developed on various software frameworks such as PyTorch, Keras, TensorFlow, and Darknet. Besides, ANNs can be conveniently deployed on dedicated hardware such as the GPU and even some ASIC accelerators like TPU, NPU, and so on. Thanks to these mature algorithms, software frameworks, and dedicated hardware, nowadays ANNs can outperform human beings in some tasks, such as recognizing faces and playing chess.

Inspired by the neural network of the brain, ANNs are composed of artificial neurons and connections along with weights. Unlike the neurons in the neuromorphic computing that mimic the biological neural dynamics, the artificial neurons in the ANN perform multiply-and-accumulate (i.e., all the inputs of a neuron are weighted and summed, and normally, a bias is added to this sum) and activation function (i.e., normally a nonlinear function). Thus, the model of the ANN is much simpler than that of event-based neuromorphic computing. The connections in the ANN provide the output of one neuron as an input to another neuron. Each connection is assigned a weight representing its relative importance, and these weights are learned during ANN training.

Even though ANN exploits abstract and fundamental mathematical models of the findings in neuroscience, its practical advantages have been well recognized in the machine leaching community, which drives the design of domain-specific architectures. Surveys in Schuman et al. (2017), Sze et al. (2017), and Chen et al. (2020) have provided sufficient technology backgrounds in the history of

ANN, its mathematical models, and recent development of customized computing architectures, especially for deep learning. This section, on the contrary, aims to illustrate essential design metrics for ANN architectures and selective design approaches to improve those. It is worth mentioning that as the advantages of ANN gradually thrive in various application domains, primary design considerations are subject to significant adjustment; therefore, it is currently immature to judge which metrics are foremost important for ANN architectures, in contrast to the classic design metrics of performance, power, and area for microprocessors.

Design Metrics for ANN Architectures

In this section, we review the important design metrics for ANN accelerator as illustrated in Fig. 25. Foremost, ANN architecture shares the common design metrics as conventional designs in the semiconductor industry, which is functionality and PPA (performance, power consumption, and silicon area). Accuracy acts as a specific design metric due to the fact that exactness in ANN computing is less demanded, since most ANN tasks, e.g., vision and audio are interactively judged by humans where approximated values are highly acceptable. Consequently, the designer trades off accuracy with PPA and even functionality to reduce manufacture cost and time to market. Another critical design metric is ecosystem. Since ANN accelerators are still in the booming phase, programming framework, system-level integration strategy, and open-source design templates are essential for standardization. Accelerators lacking programming support for state-of-the-art machine learning frameworks are unlikely to succeed. Least but not the last, rest metrics such as reliability, security, and support for design space exploration (DSE) are mostly at the research stage and will step into industry-level designs in the near future.

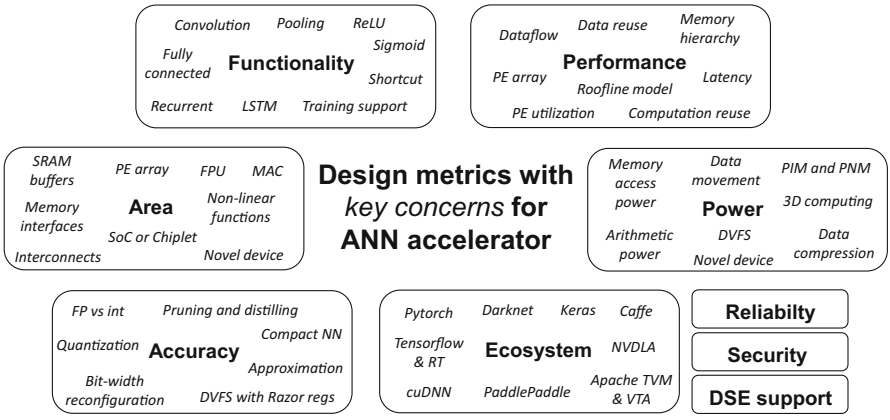


Fig. 25 Design metrics and associated key concerns for ANN accelerator

We briefly illustrate primary design metrics and associate design concerns in the following.

Functionality Current ANN architectures mainly accelerate either the inference or training phase of artificial neural networks. While training architectures tend to support backpropagation as a de facto design principle, inference accelerators are more diverse and application-specific. Neural networks are realized through layer-wise concatenation of various algorithmic kernels, which are composed of but not limited to convolution, pooling, recurrent, activation, shortcut, normalization, and fully connected kernels. For each kernel, there may exist several variations. For instance, the convolutional kernel is evolving into conventional, point-wise, depth-wise, atrous, 1D-, 2D-, and 3D convolution kernels. Pooling is evolving into max, min, global average pooling kernels. RELU and leaky RELU activation functions are common for convolutions where sigmoid and tangent are largely adopted for recurrent kernels. Therefore, it is usually impractical for a dedicated architecture to support all available and even evolving kernels. ANN architectures such as TPU adopt customized instruction-set architecture to achieve programmability and computational intensity. GPU, on the other hand, achieves high flexibility through assembly-level CUDA language and high compilation effort, but to some extent sacrifices performance and power consumption.

Performance Performance is one of the primary design considerations for all computing systems. ANN architecture is evaluated with the term of operation per second (OPS), which mainly counts for the number of multiply and addition operations. OPS can be classified as peak or average where the peak performance is straightforwardly calculated by the number of multiplier and adders in the design multiply with the maximal clock frequency. However, the average performance hardly approaches the peak one due to the low resource utilization of the processing elements (PEs). Various dataflow techniques have been proposed to increase PE utilization which targets case-by-case optimization for different kernels. Using the well-known roofline model (Williams et al. 2009), performance can be relatively fairly evaluated than counting OPS. In practical, we observe diverse performance on the same architecture when benchmarking different arithmetic kernels (Jouppi et al. 2017).

Due to the large size of ANN coefficients, DRAM is inevitable for industry-level ANN circuits. Off-chip DRAM access incurs significantly longer latency than on-chip SRAM access; therefore, appropriate management of main memory access is essential for improving performance. However, data access patterns in most kernels of neural networks are irregular for high-speed streaming-based communication protocol such as AXI, therefore affecting the efficiency of data supply which finally impairs PE utilization. Memory access optimization involves cross-layer methodologies from algorithm to circuits. Strategies have been proposed to reduce DRAM access by reusing computation based on temporal and spatial localities, which are largely observed in vision-based CNN networks.

Adopting a high-performance ANN accelerator does not imply a low latency for handling the ANN task. The latency of the neural network may only constitute a few percentages of the total latency of the target ANN task. For instance, most vision tasks involve pre- and post-processing besides the network itself, which constitute a significant portion of control heavy and floating-point operations. Such extra processing is usually executed on CPUs to explore their flexibility; however, it easily becomes the performance bottleneck of most ANN tasks. The system designer should not ignore any parts of the target application and is suggested for complete system-level modeling before digging into architecture and circuits.

Power Consumption Power consumption is critical for embedded devices and mission-critical applications. Current ANN accelerators are reported to consume a wide range of power numbers from μW to 100+ W. Power is mainly dissipated on both logic and memory access; therefore, it would be helpful to benchmark the integrated power through time, also named as energy, for various operations. For 14 nm CMOS technology, the estimated numbers are present in Dally (2021) and shown in Table 1. It is indicated that for the same data granularity of 32 bits, global memory access incurs 400x more energy than local memory access, where, on the other hand, 8-bit integer add operations take only less than 1% of the energy for local memory access of 32-bit word.

Consequently, main memory access is expensive in power consumption and, therefore, must be once again carefully managed across design abstractions. The algorithm designer favors compression of the ANN model by pruning, distilling, and quantization techniques, which may lead to an SRAM dominant design, but somehow impairs the accuracy of the ANN task and usability as well. The architect can improve operational intensity (Williams et al. 2009), which is defined as the average number of computations per fetched data, by carefully introducing buffers to save repeated data access.

Low-power techniques for ANN architectures are concerned with both circuit-level and architecture-level techniques. Circuit designers can take advantage of conventional low-power techniques, such as skillfully performing placement and routing, exploiting clock and power gating, and utilizing dynamic voltage and frequency scaling (DVFS) to optimize power. In parallel, architects have reach the consensus that reducing data movement is the key to save power, therefore novel design paradigms such as processing-in-memory (PIM) and processing-near-

Table 1 Benchmarking of energy consumption for 14 nm technology (Dally 2021). (Courtesy of Prof. Bill Dally from NVIDIA Corporation)

Memory access		**Arithmetic** FP64 FMA	5pJ
Gobal memory	640pJ/word	FP32 FMA	1.2pJ
On chip comm.	3.2pJ/word-mm	Int16 mul	260fJ
Local memory	1.6pJ/word	Int8 add	10fJ
Energy numbers for 14 nm. A word is 32 bits			

memory (PNM) have been demonstrated, where computing and storage are spatially adjacent; therefore, less data movement can be realized. For instance, PNM style heterogeneously integrated design, also known as 3D integration, has appeared in industry accelerators. Novel device such as memoristor and spintronics leads to huge power saving but is currently facing programmability support.

Silicon Area The data-intensive nature of ANN implies a chip design style filled with thousands of PEs and associated buffers. Accordingly, PEs and buffers can easily take up 90% of the silicon area. Since hardware implementation of large bit-width multiply-and-accumulation and floating point algorithmic and nonlinear activation functions are very costly in area. Optimization techniques should consider whether a huge amount of PEs should be designed uniformly or asymmetrically. An asymmetric design style can greatly reduce area cost however introducing controlling complexity. Furthermore, to achieve enough output accuracy, various neural networks have different quantization strategies. Symmetric but parallel data pipelines are inclined to occupy more than necessary silicon resources since lower-bit data representations can usually be sufficient. On the contrary, the topology and size of buffers should be carefully allocated, since small on-chip buffers can have difficulty in holding minimal demanded activations for large kernels, while huge buffers are anyway not recommended for cost in fabrication. Novel devices, such as embedded DRAM, 3D IC, and ReRAM would be essential solutions to achieve compact and cost-effective design shortly.

From the system-level perspective, typical ANN engines have become huge in size (James et al. 2020) which suffer both economically and also in terms of yield. Current system-on-chip implementations are hardly adaptive to such scaling trend. For domain-specific architectures, chiplet is highly expected to become the future solution of system integration, where industry standards for inter-chiplet communication are worth looking forward to.

Accuracy One important difference between ANN architectures and the ones in other domains is the approximation property of neural networks. Regression and classification, which are two essential types of NN outputs, both tolerate variations of intermediate computations within network layers, especially for classification. Therefore, state-of-the-art inference accelerators adopt low-cost fixed-point computation and data representation to trade-off accuracy for silicon area. However, such approximation in computing introduces significant design risks, where the functional correctness of the architecture alone does not guarantee any output accuracy. Although in reported ANNs targeting vision-based tasks, it is claimed to have sufficient accuracy by employing an 8-bit integer for intermediate computation, this is not the rule of thumb for evolving ANNs and larger images. Several kernels, such as shortcut and route, demand the merging of at least two layers with different data representations. The careless handle of such kernels can cause a huge accuracy drop. Recurrent kernels such as GRU and LSTM have a high requirement in precision due to a large number of network coefficients and sigmoid/tangent activation functions, which makes small bit-width fixed-point data representation

dispensable. Furthermore, accuracy is often used to be traded-off for performance, e.g., using smaller bit representation for performance boosting, and for power, e.g., aggressively lowering driving voltage while using Razor registers to perform error detection. Consequently, accuracy exploration plays a unique role for ANN architectures and demands dedicated support in the toolchain, for instance, precision simulator and quantization trainer, which increases the effort for software-hardware codesign. Techniques for dynamic bit-width adaption are introduced to reconfigure computational data paths for various bit-width requirements, where area overhead demands for hands-on optimization.

Ecosystem It is hard for any new accelerator to achieve public popularity. The challenge not only lies in its functionality and PPA but also mostly in the compatibility of its tool kits to the existing software and hardware frameworks, which constitutes the ANN's ecosystem. Currently, machine learning algorithms are developed and trained through mainstream frameworks such as Pytorch, Tensorflow, Keras, and Caffe. For on-site deployment, Darknet, Tensorflow RT, cuDNN, and PaddlePaddle are adopted to target various computing devices. To bridge the gap between software framework and physical devices, the Apache TVM compiles various algorithm descriptions for CPUs, GPUs, and accelerators. From hardware perspective, Nvidia's NVDLA is the first opensource machine learning accelerator which has become the initial RTL template for many industrial designs. The Versatile Tensor Accelerator (VTA) is an open, generic, and customizable deep learning accelerator with a complete TVM-based compiler stack. It is reasonably anticipated that any ANN accelerator lacking compatibility to mainstream software frameworks can hardly survive, while conservatively following the existing open design templates will hinder novelty and breakthrough.

Other Metrics Reliability and security are gradually becoming the research hotspots for ANN architectures as the application domains of ANN stepping into mission-critical systems. For instance, ANN accelerators used in airplanes and aerospace demand fault-tolerant to both transient errors and permanent failures due to highly radiated environments and harsh temperatures. Personalized ANN models must be guarded against hacking ideally by intrinsic security means in the AI circuit. The inevitable adoption of ANN in autonomous driving put foremost design consideration on accelerator safety. Research on the above directions is thriving rapidly. Current ANN execution models and architectures may possess potential weaknesses, therefore are subjected to fundamental design updates. Furthermore, high reliable and secure VLSI designs are extremely valuable and play key roles in the military and national defense. It is expected that design prototypes targeting improvements on reliability and security keep appearing soon. Tools for design space exploration such as engine for neural architecture search (NAS) are also well researched, which intend for fast identification of ANN structure for specific application and dataset and are possibly followed by automating the hardware implementation process such as software toolchain and HDL generation.

Design Abstractions and Trade-Offs

Although ANN hardware architectures are the main focus of the chapter, a successful ANN system demands a complete design flow involving cross-layer abstractions, as illustrated in Fig. 26.

Application Level Design: The design flow of the ML system originates from the application level, where the customer or designer provides the application (e.g., image classification and object detection (Iandola et al. 2016; Wu et al. 2017)) and fixes the constraints (Joulin et al. 2017) (e.g., accuracy, latency, and power consumption). Although ANN achieves significant performance for selective application kernels, it cannot be replaced by traditional rule-based algorithms in rest scenarios. The complete application is usually decomposed into subtasks, where ANN targets a few of them.

ANN Architecture Design: The ANN model is construct (Krizhevsky et al. 2012; Iandola et al. 2016) through structural exploration and parameter training. ANN structure is illustrated by its amount of layers and neurons per layer, operation types, and interconnects of the individual layer. Network parameters or weights need to be adjusted through training algorithms (e.g., back-propagation). Various template ANN models can be used as reference designs, such as Yolo-series for object detection, MobileNet as classification, and LSTM as voice recognition.

ANN Optimization: The default ANN models are usually large in size of parameters, long in execution latency, or high in power consumption and are therefore

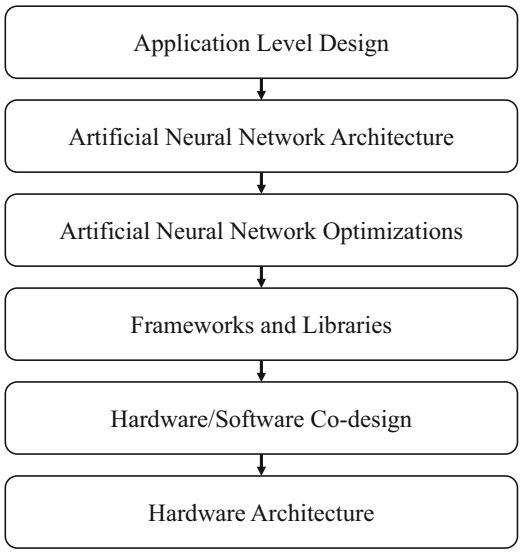


Fig. 26 Design abstractions of ANN architectures

hard to meet design constraints. Network optimization techniques have been heavily researched and practiced to reduce model size. For instance, weight pruning is adopted to compress the ANN structure such as the number of layers, neurons, and channels (Sanh et al. 2020), while quantization (Iandola et al. 2016) is used to shorten the width of data representation. State-of-the-art ANN optimization techniques can achieve 2-orders of magnitude reduction in model size with negligible accuracy loss.

Frameworks and Libraries: After the ANN models are determined and optimized, they are deployed onto physical devices through various machine learning frameworks such as Tensorflow and Pytorch. System-level libraries are leveraged to improve performance on heterogeneous computing architectures, e.g., GPU cuDNN acceleration libraries (Chetlur et al. 2014). Frameworks and libraries are usually built and integrated as part of the operating system, which bridges the application with low-level machine languages.

Hardware Software Co-design: Similar to the conventional computing architectures such as CPU and GPU, ANN architectures also have their instruction set architecture (ISA), which serves as the interface between software and hardware. Most ML architectures take advantage of CISC style ISA, where single instruction describes one specific network layer, instead of atomic instructions such as load, store, and jump in RISC-style ISA. HW/SW codesign reflects the procedure of designing ISA and translating it into the control and data flow in hardware architecture.

Hardware Architecture Design: The bottom design abstraction is the hardware architecture. Regular architect refers to prominent designs such as Google TPU and Nvidia NVDLA as initial templates. However, for ANN architects, it is essential to understand both the algorithmic kernels and architectural considerations. As ANN kernels are continuously evolving, it is practicable to decompose new kernels into a combination of existing ones to save physical overheads. Furthermore, architects are supposed to have circuit-level experiences, since large ML architecture will face difficulties in placement and routing; therefore, appropriate hierarchical implementation choices have to be made if necessary.

ANN system designers should understand that enormous gains in design metrics can be envisioned across design abstractions. At each abstraction level, however, improvements vary significantly in terms of latency/operations energy and memory, as summarized in Table 2. We can see that at the application level, by changing application characteristics, the improvement can reach up to 1000× even with the same underlying ANN models. For ANN model architectures, one can improve by the low end of the spectrum (i.e., ∼4×) with negligible accuracy loss and achieve significant gains (i.e., ∼50×) when allowing little accuracy impact (<4%). While fixing the application constraints and ANN model architectures, numerous optimization techniques can be applied to dramatically improve key parameters with

Table 2 Improvements at different ANN design abstraction levels (Keutzer)

	Latency or Ops	Energy	Memory
Application Level (Iandola et al. 2016, 2020; Wu et al. 2017, 2018, 2019; Kim et al. 2021b; Zhai et al. 2020; Joulin et al. 2017)	$10-1000\times$	$10-1000\times$	$10-1000\times$
ANN Architecture (Krizhevsky et al. 2012; Iandola et al. 2016, 2020; Sandler et al. 2018; Cai et al. 2019; Molchanov et al. 2021; Tay et al. 2020; Devlin et al. 2018; Sun et al. 2019, 2020)	$4-50\times$	$4-50\times$	$4-8\times$
ANN Optimizations (Krizhevsky et al. 2012; Iandola et al. 2016; Kim et al. 2021a; Sanh et al. 2019, 2020)	$4-7\times$	$4-16\times$	$4-8\times$
Frameworks and Libraries (Chetlur et al. 2014; Chen et al. 2018)	$3-10\times$	$3-10\times$ estimated	$3-50\times$
HW/SW Co-design (Moons et al. 2017)	$2-20\times$	$2-20\times$	(area) $4-40\times$
HW Architecture	$2-10\times$	$2-10\times$	(chip area) $2-4\times$

$<0.5\%$ impact on accuracy. Frameworks and libraries provide flexible yet powerful support to different architectures, which result in enormous improvements with $<0.5\%$ accuracy impact. By closely co-designing hardware and software, one can get improvements without compromising accuracy, e.g., reducing $24\times$ gate-count by replacing 32-bit FP MACs with int4 MACs. Even pure hardware architecture design can gain improvements by $2-10\times$.

In various scenarios, one can gain significant improvements by combining techniques at different abstraction levels. For example, the customer has determined the application and ANN model. In this case, one can achieve $12-70\times$ improvement with ANN optimizations and frameworks. If the accuracy requirements are relaxed, another $4-50\times$ improvement can be gained.

Selective ANN Architectures and Circuits

DianNao Series (Chen et al. 2014a): DianNao ("Computer" in Chinese) is well recognized for its initial design prototype which opens up huge research bodies for ANN computing architecture. The idea got inspired by previous work in Chakradhar et al. (2010) and Temam (2012) which start accelerating only small-scale multilayer perceptrons and neural networks. DianNao, in contrast, is built for deep neural networks with convolution and pooling kernels. It introduces the fundamental architecture template for ANN, which constitutes buffers of input neurons (NBin), output neurons (NBout), and synaptic weights (SB) to handle the execution resource mapping of large-scale ANNs, which has been utilized for the design reference for years. Besides, it exploited fixed-point quantization and interpolated nonlinear functions, which has proved trivial accuracy degradation for classification tasks.

Chen et al. (2014b): The next work in the DianNao series is DaDianNao ("Big computer" in Chinese), which uses DianNao as submodules to build an on-chip supercomputer. The bottleneck of design scaling into a supercomputer is the main memory access, where DaDianNao explored the possibility of localized storage of ANN weights and activations into on-chip eDRAMs, which breaks the limitation of memory bandwidth. The design is organized in a two-level hierarchy of tile and node. Each tile contains the reproduction of DianNao named as neural functional unit (NFU) and four eDRAM banks, whereas the node contains 16 tiles connected by an eDRAM router. DaDianNao proposed an aggressive architecture redesign for ANN and foresaw the potential of performance scaling (450.65× over K20M GPU) if the memory wall would have been well addressed through advanced memory technology.

Du et al. (2015): Afterward, ShiDianNao ("Vision computer" in Chinese) integrated the NFU into the pipeline of the image processor to deploy CNN on the camera. The incentive of the work was the elimination of costly DRAM access, which seemed surprising but was practically realizable for benchmarked small-scale neural networks. Variations of the dataflow for various kernels have been designed for increased energy efficiency.

Liu et al. (2015): The last member in the DianNao family PuDianNao ("Prevalent computer" in Chinese) targeted to increase the supported machine learning primitives beyond ANN including k-nearest neighbors, k-means, support-vector machine (SVM), and others. To achieve this, key computational tasks and locality properties have been extracted for all ML primitives, which guides the augmentation of NFU into a machine learning unit (MLU) with six pipeline stages. Instruction set architecture (ISA) was carefully designed to support different ML tasks.

Cambricon Series (Liu et al. 2016): After the commercialization of the DianNao series, the original DianNao team formally proposed Cambricon as an ISA for neural networks and demonstrated it on its IP named as Cambricon-ACC. The motivation was that a variety of and increasing NN kernels had resulted in a huge expansion of the instruction set, therefore causing significant physical burdens for the decoder. Inspired by the RISC ISA design principles, Cambricon decomposed complex instructions (network layers) into shorter ones, which increases the programming flexibility tremendously. With four types of instructions in the ISA, which are control, data transfer, computational and logical, a large variation of ANN tasks, e.g., LSTM, autoencoder, and restricted Boltzmann machine can be straightforwardly programmed and deployed.

Zhang et al. (2016): As the sizes of NN increased from 650-kilo neurons in AlexNet (Krizhevsky et al. 2012) to 10 billion in Coates et al. (2013), intensive computation and memory accesses have been incurred which makes efficiently processing of state-of-the-art NN on conventional accelerators a challenging problem. Algorithm designers attempted to optimize NN size through pruning and distilling, which resulted in sparse neural networks. However, NN accelerators such as DianNao fail in efficiently processing sparse networks due to missed architecture supports. To address this, Cambricon-X introduced an efficient indexing

module (IM) for selecting and transferring only needed neurons from centralized neuron buffers with reduced bandwidth requirement. Taking advantage of IM, each PEs stored irregular and compressed synapses for local computation in an asynchronous fashion, which speeds up the processing of sparse NN.

Zhou et al. (2018): While Cambricon-X handled static synapse sparsity (SSS), it did not provide architecture supports for static neuron sparsity (SNS) and dynamic neuron sparsity (DNS) which incurs large physical cause due to the IM module. To improve this, Cambricon-S aimed at alleviating the irregularity of sparse networks through a cooperative software/hardware approach. The key observation for software-based network pruning is that larger weights after training tend to gather into small clusters, which is called "local convergence." Taking advantage of such phenomenon, irregularities in a sparse network could be greatly reduced and achieve a high compression ratio of 79× for AlexNet. Afterward, hardware modules such as neuron selector (NSM) and synapse selector (SSM) handle the remaining irregularity. Other architectures handling network sparsity can be referred to in Han et al. (2016) and Albericio et al. (2016).

Zhao et al. (2019): As machine learning became pervasive in both domains of embedded and high-performance computing, it was inevitable to build multi-instance and many-instance machines based on the baseline Cambricon module. However, increased parallelism usually came along with increased programming complexity, which has been previously seen through complicated CPU and GPU APIs. To increase programming productivity, Cambricon-F introduced a fractal von Neumann architecture to iteratively manage its components. Specifically, the sub-nodes of Cambricon-F were still Cambricon-F with the same architecture and ISA; therefore, multiple hierarchies of Cambricon-F shared the same software stack thus alleviating the programming complexity. To demonstrate the advancements in architecture, two Cambricon-F instances with different scales, i.e., F100 and F1, were benchmarked with 1080Ti and DGX-1 GPUs through the famous Roofline model (Williams et al. 2009).

Google TPU (Jouppi et al. 2017): Unlike the Cambricon series seeking increased programmability, Google's Tensor Processing Unit (TPU) adopted custom ASIC design styles to increase performance. Out of three generations of TPUs, only the evaluation of the first generation was reported in detail. TPU-I targeted acceleration of NN inference and leveraged 64K into systolic array multipliers and 24 MB on-chip SRAM buffers, which in total accounted for 53% of the die silicon area. It connected with the host CPU through the PCIe 3.0 bus and utilized DDR3 DRAM for off-chip main memories. TPU-I was designed with a peak throughput of 92 TOPs per second; however, the measured performance is far less than the peak one. It was evaluated that among various machine learning tasks, on average 23% of the MACs were used which only gives a 21.4% of the peak throughput. The reason for such low MAC utilization is at least two folds. First, the bandwidth of DDR3 DRAM is quite low for such a server-level ASIC which significantly impacts the speed of data supply. Second, various neural networks have different access patterns for both activations and weights, which causes data access irregularities and further prolongs

the duration for memory access. Admitting that memory bandwidth limits the performance, TPU-II was released in 2017 with High Bandwidth Memory (HBM) technology and targeted both inference and training tasks by supporting floating-point operations. The average performance is 45 TFLOPS and can be arranged into four-chip modules with 180 TFLOPS. Of these modules, 64 are assembled into a 256-chip pod with a performance of 11.5 PetaFLOPS. Afterward, TPU-III was announced in 2018 with twice the performance of a single chip regarding TPU-II, while the deployable chips in a pod is four times the size, which gives 8× performance boosting. Details on TPU-II and TPU-III are published in Norrie et al. (2020).

Eyeriss Series (Chen et al. 2016): Architecture papers for DianNao and TPU do not provide implementation details. However, innovations in implementation are essential for energy-efficient design. Dataflow exploration is one of the key issues in ANN design to improve energy efficiency, where Eyeriss achieves high OPS/W through its well-known row stationary (RS) dataflow for the convolutional neural network. RS is designed for system-level energy optimization, especially for minimizing DRAM access with a four-level memory hierarchy to exploit data locality. Three forms of data reuse are maximized: convolutional reuse, filter reuse, and Ifmap reuse. The architecture also exploits statistics of feature maps, which involves a run-length compression (RLC) codec for compressing zero values and therefore reducing the amount of DRAM access. The data delivery network-on-chip (NoC) cooperates with RS dataflow for dynamic data gating, which achieves single-cycle data delivery to multiple destination PEs and saves the dynamic power of turned-off PEs. The innovative designs of Eyeriss also include distributed control logic and local storage inside PE. Furthermore, the data delivery NoC cannot adapt to support both high-data reuse and high-bandwidth scenarios, which was addressed in the team's follow-up work in Chen et al. (2019). Other works for optimizing ANN dataflow are referred to in Lu et al. (2017) and Kwon et al. (2018).

Thinker Series (Yin et al. 2017): Many prototypes such as Eyeriss are designed and optimized for convolutional kernels. However, other kernels such as recurrent, full connection, and scalar operations dominate in computation for networks of non-vision applications. The Thinker series chips target dynamic reconfiguration through coarse-grained reconfigurable architecture (CGRA). Thinker-I leverages output stationary dataflow for hybrid neural networks, targeting all convolutional, recurrent, and FC kernels. It introduces heterogeneous PE arrays, with general PEs for MAC functions and super PEs for MAC, pooling, activation, and scalar operations. PEs support bit-width adaptive computing for both activations and weights. On-demand array partitioning (ODAP) is proposed to process hybrid networks in parallel, thus increasing resource utilization. A multibank memory system employs a fused pattern-based memory banking strategy to exploit data reuse and reduce redundant memory access.

Yin et al. (2018a): An energy-efficient reconfigurable processor for deep neural networks with binary/ternary weights and 1/2/4/8/16-bit activations is implemented

in 28 nm technology. To save identical and redundant operations incurred by binary/ternary weights, Total-Partial-Pixel-Summation (TPPS), Kernel-Transformation-Data-Reconstruction (KTDR), and Hybrid Load-Balancing Mechanism (HLBM) technologies are employed to improve energy efficiency, which shows 6.6x improvement over state-of-the-art works. Other works for deploying circuits with binary weights or activations are referred to in Lee et al. (2018).

Yin et al. (2018b): An ultralow-power speech recognition processor is implemented in 28 nm CMOS technology, which is based on an optimized binary convolutional neural network (BCNN). A tailored self-learning mechanism is implemented to learn the features of users and improve recognition accuracy on the fly. Measurement results show that this processor supports real-time speech recognition with a power consumption of 141 uW when working at 2.5 MHz while achieving at most 98.6% recognition accuracy.

Guo et al. (2019): Thinker-IM is the first digital-CIM (computing-in-memory) mixed processor for speech recognition. It employs 16 SRAM-CIM macros for binarized recurrent neural network (RNN) computation. Its major contributions are: (1) a novel digital-CIM mixed architecture that runs an output-weight dual stationary (OWDS) dataflow, (2) multi-bit XNOR SRAM-CIM macros and corresponding CIM-aware weight adaptation, and (3) predictive early batch-normalization (BN) and binarization units (PBUs) that reduce computations in RNN. Measured results show the processing speed of 127.3us/Inference and over 90.2% accuracy. Other works for CIM-based ANN circuits are referred to in Yin et al. (2020) and Kim et al. (2019).

Types of dataflow: Since each MAC operation in the ANN typically requires three memory reads (weight, activation, and partial sum) and one memory write (updated partial sum); the bottleneck of ANN accelerators is normally in the memory access. On the other hand, the energy consumed by the data movement or the memory access is much higher than that of the computation. For example, the off-chip large memory DRAM (gigabytes) access requires up to several orders of magnitude higher energy than an ALU computation. To reduce the energy consumed by off-chip data movement, several levels of on-chip memory hierarchy have been introduced in the ANN accelerators, including the SRAM (hundred kilobytes) and register (a few kilobytes). Thus, accessing SRAM and register consumes one and two orders less energy than that of DRAM, respectively (Chen et al. 2016).

Unlike the CPU, the dataflow of the ANN accelerator is much more regular. Therefore, it is possible to design dedicated dataflow, which can leverage memory hierarchy likes DRAM-SRAM-Register to optimize for the best energy efficiency. There forms of input data local reuse (inside the PE array) opportunities exist in the ANN accelerator, i.e., convolutional reuse, feature map reuse, and filter reuse. For convolutional reuse, activations and filter weights are reused within a given channel. For feature map reuse, activations are reused across different filters. For filter reuse, the filter weights are reused across different activations. According to the data handling characteristic, the ANN dataflows in the recent accelerators can

be briefly classified into no local reuse, weight stationary, output stationary, and row stationary.

No local reuse: Even though accessing the local registers in the PE is energy efficient, they are not efficient in terms of area when compared with on-chip SRAM. While no local reuse dataflow design will increase the memory traffic as there is no data stays stationary inside the PE array. The ANN accelerator from UCLA (Zhang et al. 2015) and the DianNao (Chen et al. 2014a) are the two example designs that adopt no local reuse dataflow. In Zhang et al. (2015), the filter weights, input activations, and output partial sums are buffered using the global buffer (implemented in SRAM). In DianNao (Chen et al. 2014a), the PE array reads filter weights and input activations from the global buffer. However, special registers have been implemented to store the partial sums to reduce the energy consumed by accessing partial sums.

Weight stationary: The weight stationary dataflow refers to that the filter weights are read from DRAM into local registers and stay stationary for many MAC processing. It is able to minimize the energy consumption of reading filter weights. The weight stationary dataflow maximizes convolutional and filter reuse of weights. However, the input activations have to broadcast to all PEs, and the input and output partial sums are buffered using the global buffer. One example that implements weight stationary dataflow is the NeuFlow (Gokhale et al. 2014), where each PE has registers to keep the filter weight during processing. The input activations are broadcast to all PEs, while the partial sums are accumulated across the PEs. Some delay storage elements are required to correctly accumulate the partial sums, and thus, the required local memory is increased.

Output stationary: In order to minimize the energy consumption of partial sums' movement, output stationary dataflow has been designed in some ANN accelerators. In these accelerators, the accumulation of output partial sums stays stationary inside the PE array, while the filter weights are broadcast to all PEs and input activations are streamed across the PE array. ShiDianNao (Du et al. 2015) is a typical ANN accelerator that implements output stationary dataflow. The PEs fetch the input activations from neighboring PEs both horizontally and vertically. Delay storage elements are also required to keep data for synchronization. Global buffers are used to buffer input activations and filter weights fetching from DRAM. There are some other variants of output stationary dataflow which target the processing of convolutional layers or fully connected layers (Peemen et al. 2013).

Row stationary: Row stationary dataflow has been proposed in Chen et al. (2016) to minimize overall energy consumption, by reusing all types of data, i.e., activations, filter weights, and partial sums. Each PE is assigned to perform 1D row convolution, and the filter weights are kept stationary in the registers inside the PE. The input activations are streamed into the PE. These activations are reused thanks to their overlaps between different sliding windows. While multiple PEs can be aggregated to process 2D convolution. In the PE array, filter weights and activations can be reused across multiple PEs horizontally and

diagonally, respectively. For the partial sums, they can be accumulated across the PEs vertically, minimizing the data movement of partial sums.

Computation in 3D Memory As ANN grows larger and deeper, the overall system performance is limited by insufficient memory bandwidth and long access latency. The well-known realizations of 3D memory such as Hybrid Memory Cube (HMC) (Jeddeloh and Keeth 2012) and High Bandwidth Memory (HBM) (Lee et al. 2014) can potentially address such bottleneck due to the massive bandwidth provided by data access through parallel memory channels, named as vaults. Accordingly, a computing layer or logic die needs to be placed at the bottom layer of the 3D memory stack. Furthermore, computing architectures and programming paradigms with 3D memory need to be carefully designed.

Kim et al. (2016): Neurocube is the first architecture for ANN computing in HMC. It consists of clusters of processing engines, connected by a 2D mesh network as a processing tier, which is integrated into 3D with multiple tiers of DRAM. The PE clusters access multiple vaults in parallel. The operating principle, referred to as memory-centric computing, embeds specialized state machines within the vault controllers of HMC to drive data into the PE clusters. The paper presents the basic architecture and an analysis of the logic tier synthesized in 28 and 15 nm process technologies.

Gao et al. (2017): TETRIS is proposed to address architectural challenges provided by HMC and improve energy efficiency for a 3D memory-based ANN computing system. First, it adopted smaller on-chip buffers with different use to match the lower cost of main memory access. Second, it explored approaches to move operations closer to the actual memory locations. Third, it implemented a dataflow scheduling scheme with equivalent efficiency to the optimal schedules derived from the exhaustive search. Finally, it presented a hybrid partitioning scheme that parallelizes the NN layers across multiple vaults in the stack.

Ueyoshi et al. (2018): Although HMC and HBM provide high throughput, their access latency remains problematic and limits the performance for sparse and irregular networks. QUEST is a 3D DNN engine designed for agile random data access incorporating multi-vault SRAMs. It achieves an order of magnitude lower latency than DRAMs. The 40 nm CMOS QUEST prototype has 24 processing cores running at 300 MHz, where each core is associated with one 32b-width 4 MB SRAM vault. Inter-vault data communication is achieved through ThruChip Interface (Ditzel et al. 2014), which realizes 9.6 Gb/s/vault, combined 28.8 GB/s/module, R/W data bandwidth in a source synchronous manner.

Bit-Width Reconfiguration Requirements for numerical precision of ANN vary both across networks and among layers of the same network. Direct implementation of processing elements straightforwardly applies the worst-case data width, which incurs huge waste of computing resources for conventional networks. To optimize this, architectures with bit-width reconfiguration have been proposed to trade-off accuracy for performance and energy efficiency.

Judd et al. (2016): Stripes proposed the first ANN architecture whose execution time scales almost proportionally with the length of the used numerical representation. STR relies on bit-serial compute units and on the parallelism that is naturally present within DNNs to improve performance and energy with no accuracy loss. In addition, STR provides adaptivity enabling on-the-fly trade-offs among accuracy, performance, and energy.

Albericio et al. (2017): While Stripes tackles the statically ineffectual bits, Pragmatic's goal is to exploit both static and dynamic zero bits. It is based on the statistics that only 8% of the computation is strictly necessary for representative ANNs. Pragmatic eliminates most of the ineffectual computations on the fly by using serial-parallel shift-and-add multiplication and skipping the zero bits of the serial input. To reduce the area overhead, it incorporates several design decisions which result in a practical design.

Sharma et al. (2018): BitFusion introduces dynamic bit-level fusion/decomposition as a new dimension in the design of DNN accelerators. It constitutes an array of bit-level processing elements that dynamically fuse to match the bit-width of individual DNN layers. This flexibility in the architecture enables minimizing the computation and the communication at the finest granularity possible with no loss for accuracy. The bit-width reconfiguration leads to a large amount of area and power overhead as each BitBrick needs to have a shift logic.

Ryu et al. (2019): BitBlade presents a precision-scalable architecture that has a much smaller overhead for shift-add logic compared to BitFusion. BitBlade reduces the shift-add logic using the bitwise summation method. While each PE in BitFusion must have 16 variable shift logics, BitBlade requires only one variable shift logic for each PE. A 41% reduction in area and a 36–46% reduction in energy are evaluated compared to BitFusion.

Computation Reuse Current ANN tasks mainly target video and audio data, where such data exhibit large similarity among multiple frames, or within adjacent pixels of the same frame. Such insight implies the possibility of computation reuse, which can be used to improve energy efficiency. Furthermore, the application of reuse is not limited to the input data. Intermediate feature maps can be also reused for specific kernels such as residue layers, which reduces off-chip traffic.

Riera et al. (2018): Consecutive frames in audio and video applications demand back-to-back execution of ANN. Such inputs exhibit a high degree of similarity, causing the inputs/outputs of the different layers to be extremely similar for successive frames. It is shown that after linear quantization, 60% of the network inputs have the same quantized value. An architecture is designed to buffer the outputs of the different layers and reuse them for the next execution.

Mahmoud et al. (2018): Besides temporal similarity, spatial correlation is also found in vision applications especially for computational imaging. Diffy exploits spatial correlation to transparently reduce the number of bits needed to store the activations and the computations needed to perform. The key approach is to apply differential convolution, which operates on the delta values instead of the original activations. It boosts the performance by 7.1× over a value-agnostic accelerator.

Azizimazreah and Chen (2019): Shortcut activations for residue layers account for 4% of the total feature map data. They incur a large amount of off-chip memory access and significant power consumption. This work presents the approach that "mines" the largely unexplored opportunity of reusing shortcut and feature map data to reduce off-chip traffic. It introduces the use of logical buffers that are formed dynamically from a pool of physical buffer banks. Three procedures, namely, Prolog, Kernel, and Epilog, are proposed for collaboratively allowing shortcut data to be reused across any number of ANN layers.

Algorithm-Driver Architecture Optimization The ANN computing system can also be optimized through application-level characteristics. For instance, a modern image signal processing pipeline constitutes a sequence of IP blocks, where the output of early modules provides information on object prediction which can prevent ANN accelerators from processing frame by frame. Such guidance information differs case by case and is expected to be continuously explored in novel applications.

Buckler et al. (2018): This work uses temporal redundancy in the natural video to avoid unnecessary computation on most frames for real-time vision. An algorithm, namely, activation motion compensation (AMC), skips a series of layers in the CNN by predicting their output and then invokes the remaining layers to compute the final visual result. EVA2 architecture implements the AMC algorithm and uses an adaptive control scheme to decide which frames to run precisely. Implemented on top of a baseline ANN accelerator, the new unit reduces the average energy per frame by 54%, 62%, and 87% for three CNNs with less than 1% loss in vision accuracy.

Zhu et al. (2018): The key observation of this work is that changes in pixel data between consecutive frames represent visual motion. An algorithm is designed to leverage this motion information to relax the number of expensive CNN inferences. The key to the architectural augmentation is to co-optimize different SoC IP blocks in the vision pipeline collectively. Specifically, motion data that is naturally generated by the image signal processor (ISP) is exposed early in the vision pipeline to the CNN engine. Measurement and synthesis results show that Euphrates achieves up to 66% SoC-level energy savings.

Low Latency Inference ANN architectures increase performance through parallelization, e.g., using multiple instances of accelerators to process multiple frames simultaneously. However, this does not reduce the processing latency of a specific frame. The latency of the ANN system is comprised of multiple stages, e.g., pre- and post-processing and the actual network execution, which demands system-level inspection and joint optimization. Architectural techniques for reducing inference latency are essential for the next-generation autonomous system.

Wei et al. (2018): Tile-grained pipeline architecture (TGPA) is proposed for low latency inference of CNN models, which adopts a heterogeneous design on FPGA to support pipelining execution of multiple tiles within a single input image. By adopting a systolic array for single accelerator design and placing accelerators onto

different FPGA dies to avoid crossing-dies timing critical paths, TGPA achieves higher frequency than homogeneous designs. Experiment results show that the TGPA designs achieve up to 3× latency reduction than homogeneous designs.

Jia et al. (2020): Neural CPU is built on a binary neural network accelerator with the capability to emulate an in-order RISC-V CPU pipeline. Both the general-purpose CPU and BNN operations are realized in a single core avoiding the need for a complex communication interface. A special zero-latency transition scheme is developed to support seamless switching between CPU and BNN modes by essentially pipelining the reconfiguration, therefore significantly saving latency caused by inter-core data transfer. A two-NCPU core SoC chip is designed and fabricated using 65 nm CMOS technology.

Chen et al. (2021): Based on the observation that pre-processing and buffering procedure can take up 80% of ANN system latency, this work targets jointly optimization of three processing stages: Specifically, a frame combination module to relax the alignment incompatibility issue which reduces pre-processing time; a parallel img2col module to fetch activations for parallel computing threads, which reduces buffering time; and three modes of batch inference for various NN layers to reduce the processing time of multiple images. Up to 75% reduction in the system latency is benchmarked in selected ANNs.

Reliability and Security Nonfunctional design metrics such as reliability and security attract recent research attentions. Reliability is of prime importance for the mission-critical system; on the other hand, nanoscale CMOS circuits are reported with an increased error rate, which both motivate circuit hardening. Security covers a broad range of topics overlapping with safety and privacy. For trained ANNs with valuable models and weights, it is necessary to explore protection techniques in hardware security since deployed ANN on edge devices are difficult to be guarded solely through software methods.

Reagen et al. (2016): Minerva combines insights and techniques across the algorithm, architecture, and circuit layers, enabling low-power accelerators for executing highly accurate DNNs. It achieves high energy efficiency through fine-grain, heterogeneous data type quantization, dynamic operation pruning, and algorithm-aware fault mitigation for low-voltage SRAM operation. Specifically, by employing Razor SRAMs for fault detection and word masking and bit masking technologies for fault correction, Minerva saves 2.7× in power by aggressively scaling SRAM supply voltages.

Zhao et al. (2020): Edge devices deployed with CNN face increasing requirements for IP protection for the network models and their weights. SCA proposes a secure CNN accelerator that exploits stochastic computing to achieve IP protection at both training and inference phases. SCA exploits the precision difference between stochastic format and binary format representations as device-dependent fingerprints and integrates the fingerprints in the CNN models such that the trained model functions well only on the target devices. Furthermore, the baseline SCA is optimized with weight remapping and hybrid stochastic addition.

Bo et al. (2021): Traditional fault-tolerant design techniques heavily rely on replication for mutual verification of computing resources. However, conventional replication approaches such as modular redundancy incur huge physical overheads. OR-ML proposes an ANN architecture opportunistically exploring the chances of runtime redundancy provided by adjacent PEs in 2D PE arrays. Specifically, three protection modes, namely, mutual verify, directed verify, and self-isolate, are proposed. The architecture pipeline is augmented through low-cost chance explore, chance taken, and re-execution unit, where 30% fewer errors are observed.

Tools for ANN Design Space Exploration Designing specific architecture especially ASIC for ANN is not a trivial task that incorporates cross-layer knowledge including algorithm, architecture, and circuits. Tools for design space exploration (DSE) are intended for assisting early estimation of performance, power consumption, latency, and area cost for a given ANN and semiautomatic generation of implementable designs. DSE tools for general-purpose processors have been used for decades such as nML (Freericks 1991) and LISA (Chattopadhyay et al. 2008); similar DSE tool flows have also been recently constructed for ANN inference.

Venkatesan et al. (2019): MAGNet stands for modular accelerator generator for neural networks. It takes a target application consisting of one or more neural networks along with hardware constraints as input and produces synthesizable RTL for a neural network accelerator ASIC as well as valid mappings for running the target networks on the generated hardware. The MAGNet consists of three components: MAGNet Designer, MAGNet Mapper, and MAGNet Tuner. The MAGNet designer consists of a highly configurable architecture template with many design-time parameters, allowing the generation of DNN accelerators specialized for specific workloads and use cases. The MAGNet mapper handles the mapping of different neural networks onto the generated hardware and enables optimization of mapping strategies at run time. The MAGNet tuner uses Bayesian optimization to rapidly explore the design space and perform hardware-software co-optimization.

Xu et al. (2020): AutoDNNchip automatically generates optimized DNN accelerator implementation given the user-defined DNN models from machine learning frameworks (e.g., Pytorch), application-driven specifications (e.g., energy and latency), and resource budget (e.g., size of the processing array and memories). It consists of two integrated enablers: (1) a Chip Predictor, built on top of a graph-based accelerator representation, which can accurately and efficiently predict a DNN accelerator's energy, throughput, and latency and are based on the DNN model parameters, hardware configuration, technology-based IPs, and platform constraints; and (2) a Chip Builder, which can automatically explore the design space of DNN chips (including IP selection, block configuration, resource balance, etc.), optimize chip design via the Chip Predictor, and then generate synthesizable RTL code with optimized dataflows to achieve the target design metrics.

Architectures for Miscellaneous Networks Most ANN architectures are designed for vision tasks with conventional CNN. However, there exist rising interests to

build specific designs for other ANN networks, which are briefly summarized in the following.

Various techniques are proposed to accelerate recurrent neural networks (RNNs), such as Efficient Speech Recognition Engine (ESE) using compressed models (Han et al. 2017), a structure-based compression technique called C-LSTM (Wang et al. 2018), DeltaRNN taking advantage of delta network algorithm (Gao et al. 2018), and Efficient RNN (E-RNN) based on block-circulant matrix (Li et al. 2019a).

Reinforcement learning (RL) is widely used for decision-making in automation and robotics. Accelerators have been designed with an array of stochastic synapses feed input and the feedback path (Amravati et al. 2018), in-memory accelerator with the policy implemented on ferroelectric tunnel junction (FTJ) memristors (Berdan et al. 2019), an FPGA-based accelerator called FA3C (Cho et al. 2018), an in-switch accelerator for distributed multi-node RL (Li et al. 2019b).

To accelerate Bayesian inference, researchers have proposed FPGA accelerators with the RAM-based Linear Feedback Gaussian Random Number Generator (RLF-GRNG) and the Bayesian Neural Network-oriented Wallace Gaussian Random Number Generator (Cai et al. 2018) and parallel Gibbs sampling Markov random field accelerator (PGMA) (Ko et al. 2020).

The Capsule network is accelerated by PIM-CapsNet based on processing-in-memory (PIM) with GPU and 3D stacked technologies (Zhang et al. 2020). Super resolution (SR) acceleration is performed by Fast SR CNN (FSRCNN) consisting of seven convolutional layers and one deconvolutional layer (Dong et al 2016) and an SR-CNN processor with a global ring with four local rings and a global ring controller (Lee et al. 2019).

3D CNNs are accelerated using a template-based architecture that unifies 2D and 3D CNNs and improves the computation using uniform templates (Shen et al. 2018) and using a 3D CNN accelerator called Morph which fetches data from DRAM for parallel processing with PEs (Hegde et al. 2018).

The Graph Convolutional Networks (GCNs) have been accelerated by the following works: GraphACT, where the CPU and FPGA deals with communication-intensive and compute-intensive tasks (Zeng and Prasanna 2020); Autotuning-Workload-Balancing GCN (AWB-GCN) which partitions matrices and maps them to parallel PEs to perform column-wise-product-based Sparse-dense Matrix Multiplications (SpMMs) (Geng et al. 2020); and HyGCN with aggregation and combination engine (Yan et al. 2020) and GCNAX that fetches data into SMB and performs computations of chain SpMM (Li et al. 2021).

Attention-based tasks are accelerated by GOBO consisting of processing tiles whose data are supplied by a banked global buffer (Zadeh et al. 2020); A3 using algorithm approximation with modules of dot-product, exponent, and output (Ham et al. 2020); and FlexASR with 4 PEs and multifunction global buffer where each PE includes a weight buffer and an input buffer that sends data to perform operations of LSTM, GRU, or RNN layers (Tambe et al. 2021).

ANN Training Architectures Unlike the single-pass procedure of inference, ANN training evaluates the states of up to a billion neurons with millions of iterations.

Such computing intensity not only demands significant computing resources but also seeks high precision. Floating-point arithmetic units are essential in the process of backpropagation, where fixed-point approximation or customized data representation for training is still under research. Due to high engineering cost and design complexity, state-of-the-art training chips are mostly commercial designs from large companies, where we list renowned VLSI products from three vendors.

Gwennap (2016): Wave dataflow processor (DPU) is a training chip designed by the Wave Computing company. It is an ASIC design fabricated in TSMC 16 nm FinFET process. The Wave DPU is featured with hybrid dataflow processing architecture (combine standard instruction execution with dataflow principles) and self-timed logic (i.e., asynchronous logic) implementation. The Wave DPU consists of 1,024 clusters, and each cluster contains 16 processing elements plus other additional shared compute units. Each cluster operates asynchronously, and its peak operating frequency could be as high as 10 GHz, while a self-timed interlocking network is designed to synchronize neighboring clusters. The chip integrates four HMC interfaces (each holds 2 GB memory) and two 64-bit DDR4-2400 memory interfaces. A 16-lane PCI Express Gen3 interface is also included to connect to a host processor or network.

Yang (2019): Intel's training chip, namely, neural network processors training (NNP-T), is a direct descendent of Nervana's original ASIC design. While first-generation NNT-Ps are not productized, second-generation NNP-Ts are branded as the NNP T-1000 series and are the first chips to be productized. Fabricated TSMC's 16 nm process based on the Spring Crest microarchitecture, those chips feature several enhancements and refinements over the prior generation including a shift from Flexpoint to Bfloat16 and considerable performance uplift. Intel claims that these chips have about 3–4× the training performance of the first generation. All NNP-T 1000 chips come with 32 GB of four HBM2 stacks in a CoWoS package and come in two form factors: PCIe Gen 3 and an OCP OAM accelerator card.

Jouppi et al. (2020): The first generation of Google's Tensor Processing Unit (TPU), i.e., TPUv1, is an inference chip, while TPUv2, TPUv3, and TPUv4 are training chips. Take TPUv2, for example; each chip contains two TensorCore (not related to the Tensor Cores of NVIDIA GPUs). The TensorCore consists of an Inter-Core Interconnect, HBM, the Core Sequencer, Vector Processing Unit, the MUX that produces 32-bit FP products from 16-bit FP inputs, and the Transpose Reduction Permute Unit. Up to 256 TPUv2 chips can be connected through the 2D-torus topology to create the supercomputer for training. The peak performance of the TPUv2 chip is 46 TeraFLOPS (in 16 bits) or 3 TeraFLOPS (in 32 bits), and its thermal design power is 280 Watts per chip.

Open-Source Designs (NVIDIA deep learning accelerator (NVDLA) 2017): NVIDIA Deep Learning Accelerator (NVDLA) provides a simple, flexible, robust inference acceleration solution. It supports a wide range of performance levels and readily scales for applications ranging from smaller, cost-sensitive internet of things (IoT) devices to larger performance-oriented IoT devices. NVDLA is provided as a set of IP-core models based on open industry standards: the Verilog model is a

synthesis and simulation model in RTL form, and the TLM SystemC simulation model can be used for software development, system integration, and testing. The NVDLA software ecosystem includes an on-device software stack, a full training infrastructure to build new models that incorporate deep learning, and parsers that convert existing models to a form that is usable by the on-device software. NVDLA is open-source at https://github.com/nvdla

Moreau et al. (2019): The Versatile Tensor Accelerator (VTA) is an open, generic, and customizable deep learning accelerator with a complete TVM-based compiler stack. VTA is designed to expose the most salient and common characteristics of mainstream deep learning accelerators. Together TVM and VTA form an end-to-end hardware-software deep learning system stack that includes hardware design, drivers, a JIT runtime, and an optimizing compiler stack based on TVM. VTA is open-source at https://github.com/apache/tvm/

Xu et al. (2020): As introduced in the DSE subsection, the design exploration tool by Rice University is open-sourced at https://github.com/RICE-EIC/AutoDNNchip.git

Architectures for Classic Machine Learning

In addition to neuromorphic computing and ANNs, there exist classic machine learning approaches that are widely used. When using traditional machine learning techniques, domain experts identify low-level features and hand-engineer feature extraction to make patterns more visible to learning algorithms. Compared to multilayered ANNs, classic machine learning methods have simpler structures, and the architectures of their accelerators are thus simpler as well. The architectures of some typical traditional machine learning methods are explained in detail in the following sections:

Naive Bayes Classifier Naive Bayes Classifier (NBC) is a probabilistic classifier based on Bayes' theorem with the assumption that features are strongly independent. Visual objection recognition is performed with a simplified NBC on an FPGA (Meng et al. 2011). Its architecture is designed for a multi-class classifier on binary feature vectors, which is simple and thus uses limited hardware resources composed of several counters, one lookup table (LUT), one division operator, and a probability array storage.

Support-Vector Machine Support-vector machines (SVM) is a supervised learning method with associated learning algorithms for classification and regression tasks. SVM classification analysis is accelerated by an FPGA (Papadonikolakis and Bouganis 2012). The FPGA sends support vectors to classifier processing units composed of fixed-point and floating-point domains, where the fixed-point domain adopts parallel multipliers with a pipelined adder tree for dynamic resource usage, and the floating-point domain implements kernel functions of SVMs to produce the result via accumulation.

A parallel digital VLSI architecture is proposed to deal with SVM training and classification (Wang et al. 2014). The architecture splits data into multiple basic SVM units capable of processing variable data size with distributed cache memory. The communication is performed via a multilayer system bus that minimizes communication overhead.

K-Means Clustering K-means clustering approach partitions data points into k clusters where data points in a cluster are similar to the cluster center. A K-means clustering variant prunes the search space using a binary kd-tree data structure, and it is implemented on an FPGA (Winterstein et al. 2013). The FPGA dynamically allocates on-chip memory resources such that it reduces 70× in terms of memory compared to the worst-case of the static pre-allocation method.

Decision Tree Classification Decision tree classification (DTC) is a data-mining algorithm that partitions data by iteratively asking questions and generates a treelike model. An FPGA implementation (Saqib et al. 2013) accelerates DTC using a pipelined streaming architecture with double-buffered input and output memories to simultaneously buffer and process data.

Principal Component Analysis Principal component analysis (PCA) is a technique for dimensionality reduction of the feature space in data analysis. A reconfigurable architecture is presented for the covariance-based PCA on an FPGA (Korat and Alimohammad 2019), which is vector based and is used for both the training and mapping phases. The architecture corresponds to the block diagram of the PCA algorithm; i.e., in the training phase, it is composed of modules of zero-mean centering, covariance matrix computation, eigensolver, and sorting; in the learning phase, it utilizes matrix multiplication module to reduce the dimensionality.

Conclusions

With the emerging demand for not only neuroscience research but also commercial tasks like classification and recognition, various architectures for accelerating AI algorithms including neuromorphic computing/SNN and ANNs have been raised. Large-scale neuromorphic chips with complex neuronal mechanisms, such as SpiNNaker, Neurogrid, and BrainScales, have been built for simulation of the biological brain. Also, many neuromorphic chips, such as TrueNorth, Loihi, ODIN, and Tianjic, have been designed based on simplified neural and synaptic primitives to perform commercial tasks. These neuromorphic chips, built with analog circuits or synchronous or asynchronous digital circuits, can achieve extremely low-power consumption thanks to the event-driven characteristic of neuromorphic computing. In terms of algorithm, software framework, and dedicated hardware, real-valued-based artificial neural networks with simple models are more practical than neuromorphic computing. When designing an architecture for ANNs, several design metrics have to be considered, including accuracy, performance, power

consumption, silicon area, reliability, and security. DianNao is an initial design prototype for ANN computing architecture; Google's TPUs are designed for high training and inference performance; Eyeriss chips achieve high energy efficiency; and Thinker series mainly target RNN applications.

Due to a large number of research works, this chapter only provides a brief overview of the state-of-the-art ML accelerator architectures for neuromorphic computing and ANNs. The future edition of this chapter will constantly increase state-of-the-art amendments.

References

Akopyan F, Sawada J, Cassidy A, Alvarez-Icaza R, Arthur J, Merolla P, Imam N, Nakamura Y, Datta P, Nam GJ, Taba B (2015) Truenorth: design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. IEEE Trans Comput-Aided Des Integr Circuits Syst 34(10):1537–1557

Albericio J, Judd P, Hetherington T, Aamodt T, Jerger NE, Moshovos A (2016) Cnvlutin: ineffectual-neuron-free deep neural network computing. ACM SIGARCH Comput Archit News 44(3):1–13

Albericio J, Delmás A, Judd P, Sharify S, O'Leary G, Genov R, Moshovos A (2017) Bit-pragmatic deep neural network computing. In: Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, pp 382–394

Amravati A, Nasir SB, Thangadurai S, Yoon I, Raychowdhury A (2018) A 55nm time-domain mixed-signal neuromorphic accelerator with stochastic synapses and embedded reinforcement learning for autonomous micro-robots. In: 2018 IEEE International Solid-State Circuits Conference-(ISSCC). IEEE, pp 124–126

Anwani N, Rajendran B (2015) Normad-normalized approximate descent based supervised learning rule for spiking neurons. In 2015 international joint conference on neural networks (IJCNN). IEEE, pp 1–8

Azizimazreah A, Chen L (2019) Shortcut mining: exploiting cross-layer shortcut reuse in dcnn accelerators. In: 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, pp 94–105

Benjamin BV, Gao P, McQuinn E, Choudhary S, Chandrasekaran AR, Bussat JM, Alvarez-Icaza R, Arthur JV, Merolla PA, Boahen K (2014) Neurogrid: a mixed-analog-digital multichip system for large-scale neural simulations. Proc IEEE **102**(5):699–716

Berdan R, Marukame T, Kabuyanagi S, Ota K, Saitoh M, Fujii S (2019) In-memory reinforcement learning with moderatelystochastic conductance switching of ferroelectric tunnel junctions. In: Proceeding Symposium on VLSI Technology, pp 22–23

Bi GQ, Poo MM (1998) Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type. J Neurosci 18(24):10464–10472

Bo D et al OR-ML: enhancing reliability for machine learning accelerator with opportunistic redundancy. In: 2021 IEEE Design, Automation and Test in Europe Conference (DATE) (2021)

Bohte SM, Kok JN, La Poutre H (2002) Error-backpropagation in temporally encoded networks of spiking neurons. Neurocomputing 48(1–4):17–37

Brader JM, Senn W, Fusi S (2007) Learning real-world stimuli in a neural network with spike-driven synaptic dynamics. Neural Comput 19(11):2881–2912

Buckler M, Bedoukian P, Jayasuriya S, Sampson A (2018) EVA2: exploiting temporal redundancy in live computer vision. In: 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA). IEEE, pp 533–546

Cai R, Ren A, Liu N, Ding C, Wang L, Qian X, Pedram M, Wang Y (2018) Vibnn: hardware acceleration of Bayesian neural networks. ACM SIGPLAN Not 53(2):476–488

Cai H, Gan C, Wang T, Zhang Z, Han S (2019) Once-for-all: train one network and specialize it for efficient deployment. arXiv preprint arXiv:1908.09791

Chakradhar S, Sankaradas M, Jakkula V, Cadambi S (2010) A dynamically configurable coprocessor for convolutional neural networks. In: Proceedings of the 37th Annual International Symposium on Computer Architecture, pp 247–257

Chattopadhyay A, Meyr H, Leupers R (2008) LISA: a uniform ADL for embedded processor modeling, implementation, and software toolsuite generation. In: Processor description languages. Morgan Kaufmann, San Francisco, pp 95–132

Chen T, Du Z, Sun N, Wang J, Wu C, Chen Y, Temam O (2014a) Diannao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. ACM SIGARCH Comput Archit News 42(1):269–284

Chen Y, Luo T, Liu S, Zhang S, He L, Wang J, Li L, Chen T, Xu Z, Sun N, Temam O (2014b) Dadiannao: a machine-learning supercomputer. In: 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE, pp 609–622

Chen YH, Emer J, Sze V (2016) Eyeriss: a spatial architecture for energy-efficient dataflow for convolutional neural networks. ACM SIGARCH Comput Archit News 44(3):367–379

Chen T, Moreau T, Jiang Z, Zheng L, Yan E, Shen H, Cowan M, Wang L, Hu Y, Ceze L, Guestrin C (2018) TVM: an automated end-to-end optimizing compiler for deep learning. In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pp 578–594

Chen Y-H, Yang T-J, Emer J, Sze V (2019) Eyeriss v2: a flexible accelerator for emerging deep neural networks on mobile devices. IEEE J Emerg Sel Top Circuits Syst 9(2):292–308

Chen Y, Xie Y, Song L, Chen F, Tang T (2020) A survey of accelerator architectures for deep neural networks. Engineering 6(3):264–274

Chen W et al (2021) Improving system latency of AI accelerator with on-chip pipelined activation preprocessing and multi-mode batch inference. In: IEEE International Conference on Artificial Intelligence Circuits and Systems. IEEE

Chetlur S, Woolley C, Vandermersch P, Cohen J, Tran J, Catanzaro B, Shelhamer E (2014) cudnn: efficient primitives for deep learning. arXiv preprint arXiv:1410.0759

Chicca E, Stefanini F, Bartolozzi C, Indiveri G (2014) Neuromorphic electronic circuits for building autonomous cognitive systems. Proc IEEE 102(9):1367–1388

Cho H, Oh P, Park J, Jung W, Lee J (2019) Fa3c: FPGA-accelerated deep reinforcement learning. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pp 499–513

Coates A, Huval B, Wang T, Wu D, Catanzaro B, Andrew N (2013) Deep learning with COTS HPC systems. In: International Conference on Machine Learning. PMLR, pp 1337–1345

Dally B (2021) Sustainable computing via domain-specific architecture and efficient circuits. DATE Special Day on Sustainable HPC

Davies M, Srinivasa N, Lin TH, Chinya G, Cao Y, Choday SH, Dimou G, Joshi P, Imam N, Jain S, Liao Y (2018) Loihi: a neuromorphic manycore processor with on-chip learning. Ieee Micro 38(1):82–99

Devlin J, Chang MW, Lee K, Toutanova K (2018) Bert: pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805

Ditzel D, Kuroda T, Lee S (2014) Low-cost 3D chip stacking with ThruChip wireless connections. In: Proceedings of IEEE Hot Chips Symposium (HCS), pp 1–37

Dong C, Loy CC, Tang X (2016) Accelerating the super-resolution convolutional neural network. In: European Conference on Computer Vision. Springer (2016), pp 391–407

Du Z, Fasthuber R, Chen T, Ienne P, Li L, Luo T, Feng X, Chen Y, Temam O (2015) ShiDianNao: shifting vision processing closer to the sensor. In: Proceedings of the 42nd Annual International Symposium on Computer Architecture, pp 92–104

Folowosele F, Harrison A, Cassidy A, Andreou AG, Etienne-Cummings R, Mihalas S, Niebur E, Hamilton TJ (2009) A switched capacitor implementation of the generalized linear integrate-and-fire neuron. In: 2009 IEEE International Symposium on Circuits and Systems (ISCAS). IEEE, pp 2149–2152

Freericks M (1991) The nML machine description formalism. Leiter der Fachbibliothek Informatik, Sekretariat FR 5–4

Frenkel C, Lefebvre M, Legat JD, Bol D (2018) A 0.086-mm ^{2}12.7-pj/sop 64k-synapse 256-neuron online-learning digital spiking neuromorphic processor in 28-nm CMOS. IEEE Trans Biomed Circuits Syst 13(1):145–158

Friedmann S, Schemmel J, Grübl A, Hartel A, Hock M, Meier K (2016) Demonstrating hybrid learning in a flexible neuromorphic hardware system. IEEE Trans Biomed Circuits Syst 11(1):128–142

Furber SB, Galluppi F, Temple S, Plana LA (2014) The spinnaker project. Proc IEEE 102(5):652–665

Gao M, Pu J, Yang X, Horowitz M, Kozyrakis C (2017) Tetris: scalable and efficient neural network acceleration with 3d memory. In: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, pp 751–764.

Gao C, Neil D, Ceolini E, Liu SC, Delbruck T (2018) DeltaRNN: a power-efficient recurrent neural network accelerator. In: Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp 21–30

Geng T, Li A, Shi R, Wu C, Wang T, Li Y, Haghi P, Tumeo A, Che S, Reinhardt S, Herbordt MC (2020) AWB-GCN: a graph convolutional network accelerator with runtime workload rebalancing. In: 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, pp 922–936

Ghosh-Dastidar S, Adeli H (2009) A new supervised learning algorithm for multiple spiking neural networks with application in epilepsy and seizure detection. Neural Netw 22(10):1419–1431

Gokhale V, Jin J, Dundar A, Martini B, Culurciello E (2014) A 240 G-ops/s mobile coprocessor for deep neural networks. In: CVPR Workshop, pp 682–687

Guo R, Liu Y, Zheng S, Wu SY, Ouyang P, Khwa WS, Chen X, Chen JJ, Li X, Liu L, Chang MF (2019) A 5.1 pJ/neuron 127.3 us/inference RNN-based speech recognition processor using 16 computing-in-memory SRAM macros in 65 nm CMOS. In: 2019 Symposium on VLSI Circuits. IEEE, pp C120–C121

Gwennap L (2016) Wave accelerates deep learning-new dataflow processor targets 10x speedup for neural networks. The Linley MicroProcessor Report

Ham TJ, Jung SJ, Kim S, Oh YH, Park Y, Song Y, Park JH, Lee S, Park K, Lee JW, Jeong DK (2020) A$\hat{3}$: accelerating attention mechanisms in neural networks with approximation. In: 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, pp 328–341

Han S, Liu X, Mao H, Pu J, Pedram A, Horowitz MA, Dally WJ (2016) EIE: efficient inference engine on compressed deep neural network. ACM SIGARCH Comput Archit News 44(3):243–254

Han S, Kang J, Mao H, Hu Y, Li X, Li Y, Xie D, Luo H, Yao S, Wang Y, Yang H (2017) Ese: efficient speech recognition engine with sparse LSTM on FPGA. In: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp 75–84

Hegde K, Agrawal R, Yao Y, Fletcher CW (2018) Morph: flexible acceleration for 3d cnn-based video understanding. In: 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, pp 933–946

Herculano-Houzel S (2009) The human brain in numbers: a linearly scaled-up primate brain. Front Hum Neurosci 3:31

Hosomi M, Yamagishi H, Yamamoto T, Bessho K, Higo Y, Yamane K, Yamada H, Shoji M, Hachino H, Fukumoto C, Nagao H (2005) A novel nonvolatile memory with spin torque transfer magnetization switching: spin-RAM. In: IEEE InternationalElectron Devices Meeting, 2005. IEDM Technical Digest. IEEE, pp 459–462

Iandola FN, Han S, Moskewicz MW, Ashraf K, Dally WJ, Keutzer K (2016) SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5 MB model size. arXiv preprint arXiv:1602.07360

Iandola FN, Shaw AE, Krishna R, Keutzer KW (2020) SqueezeBERT: what can computer vision teach NLP about efficient neural networks? arXiv preprint arXiv:2006.11316

Indiveri G, Chicca E, Douglas RJ (2006) A VLSI array of low-power spiking neurons and bistable synapses with spike–timing dependent plasticity. IEEE Trans Neural Netw 17(1):211–221

Izhikevich EM (2003) Simple model of spiking neurons. IEEE Trans Neural Netw 14(6):1569–1572

James M et al (2020) Ispd 2020 physical mapping of neural networks on a wafer-scale deep learning accelerator. In: Proceedings of the 2020 International Symposium on Physical Design

Jeddeloh J, Keeth B (2012) Hybrid memory cube new DRAM architecture increases density and performance. In: 2012 Symposium on VLSI Technology (VLSIT). IEEE, pp 87–88

Jia T, Ju Y, Joseph R, Gu J (2020) NCPU: an embedded neural CPU architecture on resource-constrained low power devices for real-time end-to-end performance. In: 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, pp 1097–1109

Joulin A, Cissé M, Grangier D, Jégou H (2017) Efficient softmax approximation for GPUs. In: International Conference on Machine Learning. PMLR, pp 1302–1310

Jouppi NP, Young C, Patil N, Patterson D, Agrawal G, Bajwa R, Bates S, Bhatia S, Boden N, Borchers A, Boyle R (2017) In-datacenter performance analysis of a tensor processing unit. In: Proceedings of the 44th Annual International Symposium on Computer Architecture, pp 1–12

Jouppi NP, Yoon DH, Kurian G, Li S, Patil N, Laudon J, Young C, Patterson D (2020) A domain-specific supercomputer for training deep neural networks. Commun ACM 63(7):67–78

Judd P, Albericio J, Hetherington T, Aamodt TM, Moshovos A (2016) Stripes: bit-serial deep neural network computing. In: 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, pp 1–12

Keutzer K. What every NN accelerator architect should know about deep learning applications and software. In: keynote of 2021 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)

Kim D, Kung J, Chai S, Yalamanchili S, Mukhopadhyay S (2016) Neurocube: a programmable digital neuromorphic architecture with high-density 3D memory. ACM SIGARCH Comput Archit News 44(3):380–392

Kim H, Sim J, Choi Y, Kim LS (2019) Nand-net: minimizing computational complexity of in-memory processing for binary neural networks. In: 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, pp 661–673

Kim S, Gholami A, Yao Z, Mahoney MW, Keutzer K (2021a) I-bert: integer-only bert quantization. In: International Conference on Machine Learning. PMLR, pp 5506–5518

Kim S, Gholami A, Yao Z, Nrusimha A, Zhai B, Gao T, Mahoney MW, Keutzer K (2021b) Q-ASR: Integer-Only Zero-Shot Quantization for Efficient Speech Recognition. arXiv e-prints, arXiv-2103

Ko GG, Chai Y, Donato M, Whatmough PN, Tambe T, Rutenbar RA, Brooks D, Wei GY (2020) A 3mm 2 programmable Bayesian inference accelerator for unsupervised machine perception using parallel Gibbs sampling in 16nm. In: 2020 IEEE Symposium on VLSI Circuits. IEEE, pp 1–2

Korat UA, Alimohammad A (2019) A reconfigurable hardware architecture for principal component analysis. Circuits Syst Sig Process 38(5):2097–2113

Krizhevsky A, Sutskever I, Hinton GE (2012) Imagenet classification with deep convolutional neural networks. Adv Neural Inf Process Syst 25:1097–1105

Kwon H, Samajdar A, Krishna T (2018) Maeri: enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects. ACM SIGPLAN Not 53(2):461–475

Lee DU, Kim KW, Kim KW, Kim H, Kim JY, Park YJ, Kim JH, Kim DS, Park HB, Shin JW, Cho JH (2014) 25.2 A 1.2 V 8Gb 8-channel 128GB/s high-bandwidth memory (HBM) stacked DRAM with effective microbump I/O test methods using 29nm process and TSV. In: 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC). IEEE, pp 432–433

Lee J, Kim C, Kang S, Shin D, Kim S, Yoo H (2018) UNPU: a 50.6TOPS/W unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision. In: 2018 IEEE International Solid – State Circuits Conference (ISSCC), pp 218–220

Lee J, Shin D, Lee J, Lee J, Kang S, Yoo HJ (2019) A full HD 60 fps CNN super resolution processor with selective caching based layer fusion for mobile devices. In: 2019 Symposium on VLSI Circuits. IEEE, pp C302–C303

Li Z, Ding C, Wang S, Wen W, Zhuo Y, Liu C, Qiu Q, Xu W, Lin X, Qian X, Wang Y (2019a) E-RNN: Design optimization for efficient recurrent neural networks in FPGAs. In: 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, pp 69–80

Li Y, Liu IJ, Yuan Y, Chen D, Schwing A, Huang J (2019b) Accelerating distributed reinforcement learning with in-switch computing. In: 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA). IEEE, pp 279–291

Li J, Louri A, Karanth A, Bunescu R (2021) GCNAX: a flexible and energy-efficient accelerator for graph convolutional neural networks. In: 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA). IEEE, pp 775–788

Lines A, Joshi P, Liu R, McCoy S, Tse J, Weng YH, Davies M (2018) Loihi asynchronous neuromorphic research chip. In: 2018 24th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC). IEEE, pp 32–33

Liu D, Chen T, Liu S, Zhou J, Zhou S, Teman O, Feng X, Zhou X, Chen Y (2015) Pudiannao: a polyvalent machine learning accelerator. ACM SIGARCH Comput Archit News 43(1):369–381

Liu S, Du Z, Tao J, Han D, Luo T, Xie Y, Chen Y, Chen T (2016) Cambricon: an instruction set architecture for neural networks. In: 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA). IEEE, pp 393–405

Liu C, Bellec G, Vogginger B, Kappel D, Partzsch J, Neumärker F, Höppner S, Maass W, Furber SB, Legenstein R, Mayr CG (2018) Memory-efficient deep learning on a spinnaker 2 prototype. Front Neurosci 12:840

Lu W, Yan G, Li J, Gong S, Han Y, Li X (2017) Flexflow: a flexible dataflow accelerator architecture for convolutional neural networks. In: 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, pp 553–564

Maher MAC, Deweerth SP, Mahowald MA, Mead CA (1989) Implementing neural architectures using analog VLSI circuits. IEEE Trans Circuits Syst 36(5):643–652

Mahmoud M, Siu K, Moshovos A (2018) Diffy: a Déjà vu-free differential deep neural network accelerator. In: 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, pp 134–147

Martin AJ (1990) The limitations to delay-insensitivity in asynchronous circuits. In: Beauty is our business. Springer, New York, pp 302–311

Martin AJ, Nyström M (2004) CAST: Caltech asynchronous synthesis tools. In: Asynchronous Circuit Design Working Group Workshop, Turku

Mead C (1990) Neuromorphic electronic systems. Proc IEEE 78(10):1629–1636

Meng H, Appiah K, Hunter A, Dickinson P (2011) FPGA implementation of naive bayes classifier for visual object recognition. In: CVPR 2011 WORKSHOPS. IEEE, pp 123–128

Mitchell TM (1997) Machine learning. McGraw Hill. ISBN 0-07-042807-7

Molchanov P, Hall J, Yin H, Kautz J, Fusi N, Vahdat A (2021) HANT: hardware-aware network transformation. arXiv preprint arXiv:2107.10624

Moons B, Uytterhoeven R, Dehaene W, Verhelst M (2017) 14.5 envision: a 0.26-to-10tops/w subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28 nm FDSOI. In: 2017 IEEE International Solid-State Circuits Conference (ISSCC). IEEE, pp 246–247

Moreau T, Chen T, Vega L, Roesch J, Yan E, Zheng L, Fromm J, Jiang Z, Ceze L, Guestrin C (2019) A hardware–software blueprint for flexible deep learning specialization. IEEE Micro 39(5):8–16

Norrie T, Patil N, Yoon DH, Kurian G, Li S, Laudon J, Young C, Jouppi NP, Patterson DA (2020) Google's Training Chips Revealed: TPUv2 and TPUv3. In: Hot Chips Symposium, pp 1–70

NVIDIA (2017) NVIDIA deep learning accelerator (NVDLA). http://nvdla.org

Papadonikolakis M, Bouganis CS (2012) Novel cascade FPGA accelerator for support vector machines classification. IEEE Trans Neural Netw Learn Syst 23(7):1040–1052

Peemen M, Setio AAA, Mesman B, Corporaal H (2013) Memory-centric accelerator design for convolutional neural networks. In: IEEE International Conference on Computer Design (ICCD), pp 13–19

Pei J, Deng L, Song S, Zhao M, Zhang Y, Wu S, Wang G, Zou Z, Wu Z, He W, Chen F (2019) Towards artificial general intelligence with hybrid Tianjic chip architecture. Nature **572**(7767):106–111

Reagen B, Whatmough P, Adolf R, Rama S, Lee H, Lee SK, Hernández-Lobato JM, Wei GY, Brooks D (2016) Minerva: enabling low-power, highly-accurate deep neural network accelerators. In: 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA). IEEE, pp 267–278

Riera M, Arnau JM, González A (2018) Computation reuse in DNNs by exploiting input similarity. In: 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA). IEEE, pp 57–68

Ryu S, Kim H, Yi W, Kim JJ (2019) Bitblade: area and energy-efficient precision-scalable neural network accelerator with bitwise summation. In: Proceedings of the 56th Annual Design Automation Conference 2019, pp 1–6

Sandler M, Howard A, Zhu M, Zhmoginov A, Chen LC (2018) Mobilenetv2: inverted residuals and linear bottlenecks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp 4510–4520

Sanh V, Debut L, Chaumond J, Wolf T (2019) DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. arXiv preprint arXiv:1910.01108

Sanh V, Wolf T, Rush A (2020) Movement pruning: adaptive sparsity by fine-tuning. Adv Neural Inf Process Syst 33:20378–20389

Saqib F, Dutta A, Plusquellic J, Ortiz P, Pattichis MS (2013) Pipelined decision tree classification accelerator implementation in FPGA (DT-CAIF). IEEE Trans Comput 64(1):280–285

Schemmel J, Brüderle D, Grübl A, Hock M, Meier K, Millner S (2010) A e neuromorphic hardware system for large-scale neural modeling. In: 2010 IEEE International Symposium on Circuits and Systems (ISCAS). IEEE, pp 1947–1950

Schuman CD, Potok TE, Patton RM, Birdwell JD, Dean ME, Rose GS, Plank JS (2017) A survey of neuromorphic computing and neural networks in hardware. arXiv preprint arXiv:1705.06963

Sharma H, Park J, Suda N, Lai L, Chau B, Chandra V, Esmaeilzadeh H (2018) Bit fusion: bit-level dynamically composable architecture for accelerating deep neural network. In: 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA). IEEE, pp 764–775

Shen J, Huang Y, Wang Z, Qiao Y, Wen M, Zhang C (2018) Towards a uniform template-based architecture for accelerating 2D and 3D CNNs on FPGA. In: Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp 97–106

Sun Z, Yu H, Song X, Liu R, Yang Y, Zhou D (2019) Mobilebert: task-agnostic compression of bert by progressive knowledge transfer

Sun Z, Yu H, Song X, Liu R, Yang Y, Zhou D (2020) Mobilebert: a compact task-agnostic bert for resource-limited devices. arXiv preprint arXiv:2004.02984

Sze V, Chen YH, Yang TJ, Emer JS (2017) Efficient processing of deep neural networks: a tutorial and survey. Proc IEEE 105(12):2295–2329

Tambe T, Yang EY, Ko GG, Chai Y, Hooper C, Donato M, Whatmough PN, Rush AM, Brooks D, Wei GY (2021) 9.8 A 25 mm 2 SoC for IoT devices with 18 ms noise-robust speech-to-text latency via Bayesian speech denoising and attention-based sequence-to-sequence DNN speech recognition in 16 nm FinFET. In: 2021 IEEE International Solid-State Circuits Conference (ISSCC), vol 64. IEEE, pp 158–160

Tay Y, Dehghani M, Abnar S, Shen Y, Bahri D, Pham P, Rao J, Yang L, Ruder S, Metzler D (2020) Long range arena: a benchmark for efficient transformers. arXiv preprint arXiv:2011.04006

Temam O (2012) A defect-tolerant accelerator for emerging high-performance applications. In: 2012 39th Annual International Symposium on Computer Architecture (ISCA). IEEE, pp 356–367

Tuma T, Pantazi A, Le Gallo M, Sebastian A, Eleftheriou E (2016) Stochastic phase-change neurons. Nat Nanotechnol 11(8):693

Ueyoshi K, Ando K, Hirose K, Takamaeda-Yamazaki S, Kadomoto J, Miyata T, Hamada M, Kuroda T, Motomura M (2018) QUEST: a 7.49 TOPS multi-purpose log-quantized DNN inference engine stacked on 96MB 3D SRAM using inductive-coupling technology in 40nm CMOS. In: 2018 IEEE International Solid-State Circuits Conference-(ISSCC). IEEE, pp 216–218

Venkatesan R, Shao YS, Wang M, Clemons J, Dai S, Fojtik M, Keller B, Klinefelter A, Pinckney N, Raina P, Zhang Y (2019) Magnet: a modular accelerator generator for neural networks. In: 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). IEEE, pp 1–8

Wang Q, Li P, Kim Y (2014) A parallel digital VLSI architecture for integrated support vector machine training and classification. IEEE Trans Very Large Scale Integr(VLSI) Syst 23(8):1471–1484

Wang S, Li Z, Ding C, Yuan B, Qiu Q, Wang Y, Liang Y (2018) C-LSTM: enabling efficient LSTM using structured compression techniques on FPGAs. In: Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp 11–20

Waser R, Dittmann R, Staikov G, Szot K (2009) Redox-based resistive switching memories–nanoionic mechanisms, prospects, and challenges. Adv Mater 21(25–26):2632–2663

Wei X, Liang Y, Li X, Yu CH, Zhang P, Cong J (2018) TGPA: tile-grained pipeline architecture for low latency CNN inference. In: Proceedings of the International Conference on Computer-Aided Design, pp 1–8

Wijekoon JH, Dudek P (2008) Compact silicon neuron circuit with spiking and bursting behaviour. Neural Netw 21(2–3):524–534

Williams S, Waterman A, Patterson D (2009) Roofline: an insightful visual performance model for multicore architectures. Commun ACM 52(4):65–76

Winterstein F, Bayliss S, Constantinides GA (2013) September. FPGA-based K-means clustering using tree-based data structures. In: 2013 23rd International Conference on Field Programmable Logic and Applications. IEEE, pp 1–6

Wong CG, Martin AJ (2003) High-level synthesis of asynchronous systems by data-driven decomposition. In: Proceedings of the 40th Annual Design Automation Conference, pp 508–513

Wu B, Iandola F, Jin PH, Keutzer K (2017) Squeezedet: unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops, pp. 129–137

Wu B, Wan A, Yue X, Keutzer K (2018) Squeezeseg: convolutional neural nets with recurrent crf for real-time road-object segmentation from 3D lidar point cloud. In: 2018 IEEE International Conference on Robotics and Automation (ICRA). IEEE, pp 1887–1893

Wu B, Zhou X, Zhao S, Yue X, Keutzer K (2019) Squeezesegv2: improved model structure and unsupervised domain adaptation for road-object segmentation from a lidar point cloud. In: 2019 International Conference on Robotics and Automation (ICRA). IEEE, pp 4376–4382

Xu P, Zhang X, Hao C, Zhao Y, Zhang Y, Wang Y, Li C, Guan Z, Chen D, Lin Y (2020) AutoDNNchip: an automated DNN chip predictor and builder for both FPGAs and ASICs. In: Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp 40–50

Yan M, Deng L, Hu X, Liang L, Feng Y, Ye X, Zhang Z, Fan D, Xie Y (2020) HyGCN: a GCN accelerator with hybrid architecture. In: 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, pp 15–29

Yang A (2019) Deep learning training at scale spring crest deep learning accelerator (intel® nervana™ NNP-T). In: 2019 IEEE Hot Chips 31 Symposium (HCS). IEEE, pp 1–20

Yang S, Wang J, Deng B, Liu C, Li H, Fietkiewicz C, Loparo KA (2018) Real-time neuromorphic system for large-scale conductance-based spiking neural networks. IEEE Trans Cybern 49(7):2490–2503

Yin S, Ouyang P, Tang S, Tu F, Li X, Zheng S, Lu T, Gu J, Liu L, Wei S (2017) A high energy efficient reconfigurable hybrid neural network processor for deep learning applications. IEEE J Solid-State Circuits 53(4):968–982

Yin S, Ouyang P, Yang J, Lu T, Li X, Liu L, Wei S (2018a) An ultra-high energy-efficient reconfigurable processor for deep neural networks with binary/ternary weights in 28nm CMOS. In: 2018 IEEE Symposium on VLSI Circuits. IEEE, pp 37–38

Yin S, Ouyang P, Zheng S, Song D, Li X, Liu L, Wei S (2018b) A 141 uw, 2.46 pj/neuron binarized convolutional neural network based self-learning speech recognition processor in 28 nm CMOS. In: 2018 IEEE Symposium on VLSI Circuits. IEEE, pp 139–140

Yin S, Jiang Z, Seo JS, Seok M (2020) XNOR-SRAM: in-memory computing SRAM macro for binary/ternary deep neural networks. IEEE J Solid-State Circuits 55(6):1733–1743

Zadeh AH, Edo I, Awad OM, Moshovos A (2020) GOBO: quantizing attention-based nlp models for low latency and energy efficient inference. In: 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, pp 811–824

Zeng H, Prasanna V (2020) Graphact: accelerating gcn training on CPU-FPGA heterogeneous platforms. In: Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp 255–265

Zhai B, Gao T, Xue F, Rothchild D, Wu B, Gonzalez JE, Keutzer K (2020) Squeezewave: Extremely lightweight vocoders for on-device speech synthesis. arXiv preprint arXiv:2001.05685

Zhang C, Li P, Sun G, Guan Y, Xiao B, Cong J (2015) Optimizing FPGA-based accelerator design for deep convolutional neural networks. In: Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 161–170

Zhang S, Du Z, Zhang L, Lan H, Liu S, Li L, Guo Q, Chen T, Chen Y (2016) Cambricon-X: an accelerator for sparse neural networks. In: 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, pp 1–12

Zhang J, Wu H, Wei J, Wei S, Chen H (2019) An asynchronous reconfigurable SNN accelerator with event-driven time step update. In: 2019 IEEE Asian Solid-State Circuits Conference (A-SSCC). IEEE, pp 213–216

Zhang X, Song SL, Xie C, Wang J, Zhang W, Fu X (2020) Enabling highly efficient capsule networks processing through a PIM-based architecture design. In: 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, pp 542–555

Zhao Y, Du Z, Guo Q, Liu S, Li L, Xu Z, Chen T, Chen Y (2019) Cambricon-F: machine learning computers with fractal von Neumann architecture. In: 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA). IEEE, pp 788–801

Zhao L, Zhang Y, Yang J (2020) SCA: a secure CNN accelerator for both training and inference. In: 2020 57th ACM/IEEE Design Automation Conference (DAC). IEEE, pp 1–6

Zhou X, Du Z, Guo Q, Liu S, Liu C, Wang C, Zhou X, Li L, Chen T, Chen Y (2018) Cambricon-S: addressing irregularity in sparse neural networks through a cooperative software/hardware approach. In: 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, pp 15–28

Zhu Y, Samajdar A, Mattina M, Whatmough P (2018) Euphrates: algorithm-SoC co-design for low-power mobile continuous vision. In: Proceedings of the 45th Annual International Symposium on Computer Architecture, pp 547–560

Computer Arithmetic

11

Farhad Merchant

Contents

Introduction . . . 382
Definitions . . . 385
  Radix . . . 385
  Positional Notation . . . 386
  Absolute Error . . . 386
  Relative Error . . . 386
  Numerical Precision . . . 387
  Units in the Last Place . . . 387
  Machine Epsilon . . . 387
  Floating-Point Operations Per Second . . . 387
Integer Arithmetic . . . 387
  Gray Code . . . 388
  Unary Code . . . 389
Fixed-Point Arithmetic . . . 389
Floating-Point Arithmetic . . . 390
  IEEE 754 . . . 390
  Floating-Point Approximate Circuits . . . 393
  Posit Arithmetic . . . 394
  Other Formats . . . 395
Hardware Implementations . . . 396
  Adders . . . 396
  Multipliers . . . 397
  Dividers . . . 398
  Square Root . . . 398
Conclusion . . . 398
References . . . 399

F. Merchant (✉)
University of Groningen, Groningen, The Netherlands
e-mail: f.a.merchant@rug.nl

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_13

Abstract

Computer arithmetic has been an active area of research since the advent of computers. The number system study diverged to fit the underlying computer architecture and applications. The acceleration of arithmetic circuits has been a challenging task due to the complexities involved in hardware designs. The advances in technologies, backed by innovations, led to the performance improvements in the sequential arithmetic circuits until the breakdown of Moore's law. There was significant progress in the meantime in the domain of number representations to fit maximum information per bit. In the late 1990s and early 2000s, the inventions in energy and area efficient arithmetic circuits flourished with growing application requirements. The three formats, integer, fixed-point, and floating-point, became prominent based on application requirements, and several approximation techniques were exercised in the representations and across the representations. Post-2010, approximate computing rose to prominence as several applications could withstand "enough" precision and numerical accuracy of the arithmetic. While the approximate computing completely undermined the reproducibility aspects of the arithmetic, rendering high-performance gains, this chapter will discuss an overall overview of the current state and future directions for computer arithmetic and arithmetic architectures.

Keywords

Computer arithmetic · Integer arithmetic · Fixed-point arithmetic · Floating-point arithmetic

Introduction

The quest for arithmetic research begun with mechanical computers, much before the advent of digital computers. The early scientists developed mechanical calculators to perform basic arithmetic operations. *Napier's bones*, *Abacus*, and *Pascal's calculator* are the examples. Meanwhile, the binary number system was extensively studied in Europe by *Thomas Harriot*, *Juan Caramuel y Lobkowitz*, and *Gottfried Leibniz*. However, the binary number system has its roots in multiple cultures such as Egypt, China, and India. During the mid-2000, with the arrival of the *von Neumann model*, the arithmetic hardware research gained momentum. A tentative timeline of the developments is shown in Fig. 1.

Until 1970, computers used a "multiplier routine" to compute multiplications. The multiplier routine is used to shift and accumulate the partial products. The first computers that were enabled by the multiplication instruction were Motorola 6809 and Intel MCS-51 family. Later, Intel launched 8087, a floating-point coprocessor compliant to x87 (also known as *Numeric Processor eXtention* – NPX) (Palmer 1980). The 8087 coprocessor, contained in a 40-pin DIP packaged chip, was

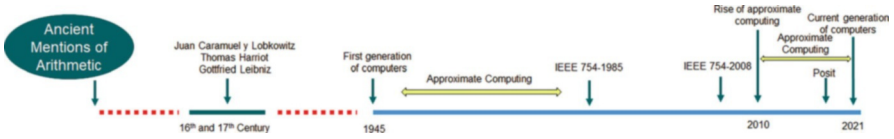


Fig. 1 Timeline of events in the domain of computer arithmetic

capable of computing addition, multiplication, subtraction, division, and square root. At 50k floating-point operations per second, the 8087 coprocessor could also compute exponential, logarithmic, and trigonometric functions. The development and popularity of the 8087 coprocessor was a significant event that motivated the need and formulation of the IEEE 754 floating-point standard that was released in the year 1985 (IEEE Standard for Binary Floating-Point Arithmetic 1985).

From Intel 80486 onward, the floating-point unit was an on-chip tightly coupled function unit, rather than a coprocessor. System and pin diagrams of 8087 are shown in Fig. 2, and the die of an unpackaged chip is depicted in Fig. 3. The 8087 coprocessor does not have a dedicated multiplier unit; rather multiplication, division, and square root operations were performed using shift-add operations. The transcendental functions such as `tan`, `arctan`, `log`, `power`, etc. are computed using CORDIC algorithms that use special constants for the shift-add operations. Though the 8087 coprocessor laid the foundation for IEEE 754 standard, the floating-point arithmetic supported by the coprocessor was not fully compliant with the IEEE standard. The 80387 coprocessor was the first-ever design that was fully compliant to IEEE 754-1985 standard based arithmetic and formats. The 80486 microprocessor was the on-chip floating-point unit integrated processor. The development of an on-chip floating-point arithmetic unit prompted innovations in the domain of area and energy-efficient floating-point architectures and software libraries such as *SoftFloat* (Garner 1976; Marasa and Matula 1973).

There have been several alternatives to the IEEE 754 compliant arithmetic and formats proposed in the literature. Logarithmic Number System (LNS), Tapered Floating-point Representation (TFR), interval arithmetic, and posit arithmetic are some of the examples (Gustafson and Yonemoto 2017; Swartzlander and Alexopoulos 1975).

The major contributions in this chapter are as follows:

- High-level details of different arithmetic formats in the play ranging from integer to some of the most recent ones
- Qualitative and quantitative comparison of the different arithmetic formats

The rest of the chapter is organized as follows: The relevant definitions in the domain of computer arithmetic are covered in section "Definitions." In section "Integer Arithmetic," integer arithmetic is covered. Fixed-point arithmetic and the implementations of fixed-point arithmetic hardware are described in section "Fixed-Point Arithmetic." In section "Floating-Point Arithmetic," floating-point

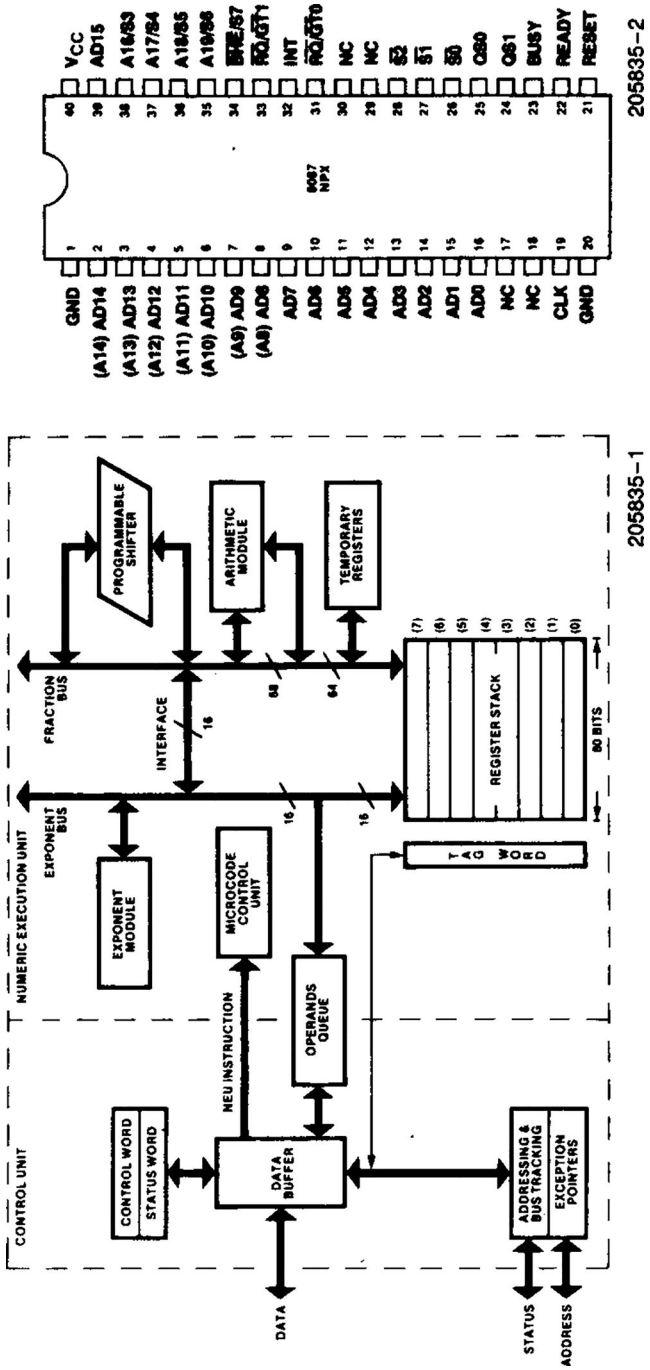


Fig. 2 8087 floating-point coprocessor. (Figure source Intel 2021)

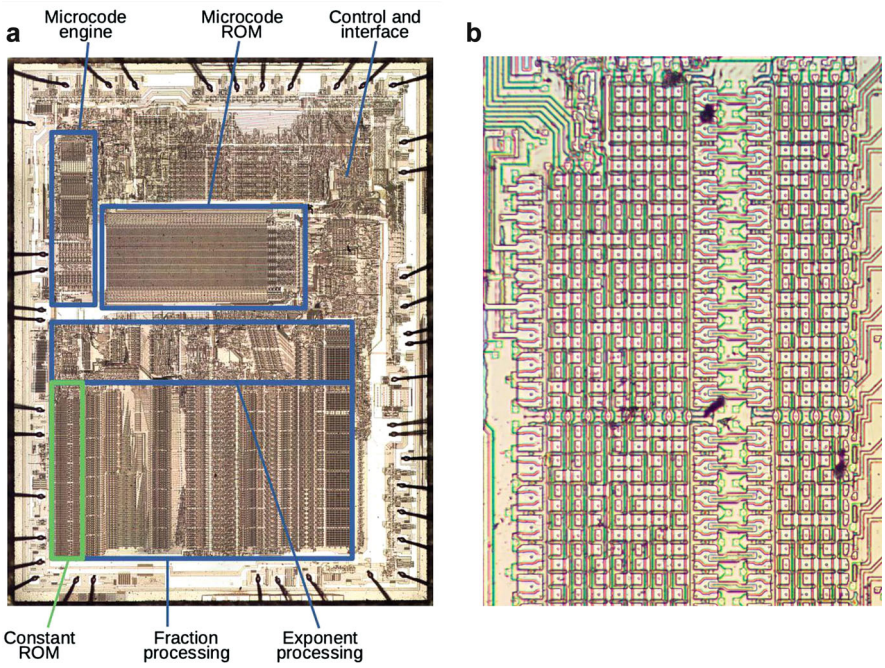


Fig. 3 Die of the 8087 coprocessor chip highlighting the main components of the design (**a**) and part of the constant ROM (**b**). (Figure source Ken Shirriff's Blog 2021)

arithmetic, different proposals in the domain of floating-point arithmetic, and its hardware implementations are discussed. In section "Hardware Implementations," an overview of hardware implementations of binary adders and multipliers is discussed. The work is summarized in section "Conclusion."

Definitions

Before delving into the arithmetic formats and their hardware implementation details, it is advisable to understand a few definitions of computer arithmetic formats, including the denotations associated with arithmetic efficiency in hardware.

Radix

Radix is defined as the base of the number system. The most prevalent radices are binary (radix 2) and decimal (radix 10). The binary numbers are understandable for the machines, and the human understanding of the binary numbers is limited. As numbers get bigger, the possibility and capabilities for humans to decode the numbers or perform any arithmetic on the number are little. The decimal numbers

are, on the other hand, easy to understand for humans. There are possibilities to have numbers with another radix than 2 for the computers, especially with the technologies that can support switching at multiple voltage levels. The discussion on multivalued logic and digital circuit design for the multivalued logic is beyond the scope of this exposition.

Positional Notation

Positional notation refers to the number representation system where each digit's position has a place value. The number is the sum of the products of each digit by its place value. A generic string is given by

$$\ldots d_3 b^3 + d_2 b^2 + d_1 b^1 + d_0 b^0 + d_{-1} b^{-1} + d_{-2} b^{-2} + d_{-2} b^{-2} \ldots \quad (1)$$

In positional notation, the radix is the base of the number system, and d ranges from 0 to $b - 1$. To separate the positive (inclusive of 0) and negative exponent, a radix point is used. The radix point concept is heavily emphasized in non-integer number representations, especially in the number system containing fractional parts.

Absolute Error

Absolute error in the arithmetic computation is defined as $|computed\ value - correct\ value|$. Often, the absolute error is misleading, especially while comparing the errors in the quantities that are in different scales. For example, for variable x, the computed value is 11, while the correct value is 10, and the absolute error is 1, and for variable y, the computed value is 101, while the correct value is 100, and the absolute error is the same as the absolute error in x. The computation of variable x is off by 10%, while y is off by 1%. In this case, the absolute error does not provide a conclusive picture of the error in the measurement.

Relative Error

Relative error is more reliable while comparing error in the measurements even at a different scale. The relative error is obtained by normalizing the absolute error by the correct value and is given by

$$relative\ error = \frac{absolute\ error}{|correct\ value|} \quad (2)$$

For the example of variables x and y in section "Absolute Error," the relative error in the computation of both the variables will be 0.1 and 0.01, respectively. A relative error has two features that should be considered in practice: (i) relative error

is undefined if the *correct value* is zero, and (ii) relative error makes sense only if the scale used has a true meaningful zero.

Numerical Precision

In the computer system, *precision* is defined as the number of digits available to accommodate the numerical value. Precision does not provide any information on the correctness of the represented number.

Accuracy (or numerical accuracy) is defined as the closeness of the measured value to the true value. Contrary to precision, accuracy provides a measure for the correctness of the measured quantity.

Units in the Last Place

Unit is the last place (ULP) is defined as the minimum distance between the two numbers in a computer system. ULP is used as a metric to identify the trustworthiness of the computer system (Goldberg 1991).

Machine Epsilon

Rounding is used for selecting the nearest approximation in the floating-point arithmetic. Machine epsilon is defined as the maximum relative error of the chosen rounding procedure.

Floating-Point Operations Per Second

Floating-point operations per second (FLOPS or flops) is one of the metrics to decide the performance of the computer. Sometimes, flops provide better information than the metric such as instructions per cycle. The flop metric is extensible to different types of computer systems, and hence, it is applied to single computers, racks, and distributed systems.

Integer Arithmetic

Unsigned integers can be captured in Eq. 1 by leaving out the digits with a negative exponent. In that case, the digits of an integer can be given by

$$\ldots d_3 b^3 + d_2 b^2 + d_1 b^1 + d_0 b^0 \tag{3}$$

The unsigned integers represented by Eq. 3 have a major drawback in representing the negative quantities. An obvious idea could be to represent the most significant digit as a sign bit. The resulting binary system can incorporate negative and positive numbers and is almost symmetric about zero. The bits after the sign bit represent the magnitude of the number. The signed-magnitude representation has multiple disadvantages: (i) redundant representation of zero (+0 and −0) and (ii) complex adder and subtraction circuits.

To overcome the issues related to signed-magnitude number system, *tow's complement arithmetic* was designed. Two's complement of a binary number is achieved by inverting all the bits and then adding one to the number. A 3-bit example is:

Decimal value	Two's complement representation
0	000
1	001
2	001
3	001
−1	111
−2	110
−3	101
−4	100

Interestingly, there is only one representation for zero in two's complement format, and addition, subtraction, and multiplication are similar to unsigned binary numbers, resulting in simpler hardware implementations. Negating a number represented by two's complement format can be attained by flipping all the bits and then adding one. Apart from the two's complement, there have been other number representations that are prevalent in several application domains.

Gray Code

Gray code is also known as reflected binary code (RBC). In Gray code, two successive numbers differ only by 1 bit. A 3-bit example of Gray code is:

Decimal value	Binary value	Gray code
0	000	000
1	001	001
2	001	011
3	001	010
4	111	110
5	110	111
6	101	101
7	100	100

The Gray codes were designed to overcome the physical limitation of the binary number system. In the binary number system, the difference between two consecutive numbers can be more than 1 bit, leading to the synchronization issue while switching. For an N-bit binary number, the maximum difference between the bits can range up to $N - 1$, while in Gray code it is always a 1-bit difference.

Due to non-simple arithmetic circuits, Gray codes have limited application. Some of the prominent applications of Gray codes are position encoders, genetic algorithms, error correction, Boolean circuit minimization, and arithmetic counters. Gray codes have also found application in low-power computer architecture bus design due to their switching characteristics.

Unary Code

Unary coding, also known as thermometer coding, is coding representing numbers in the form of ones followed by zero. An example of unary numbers is:

Decimal value	Unary code
0	0
1	10
2	110
3	1110
4	11110
5	111110

Alternatively, ones can be replaced by zeros and vice versa without the loss of generality. Unary coding has a significant disadvantage in that it is not amenable to basic arithmetic operations, and hence the adoption of unary coding in mainstream computing is undesired.

Fixed-Point Arithmetic

Fixed-point number format, as the name suggests, has a predefined position for the radix point. Fixed-point numbers are useful in representing fractional values and have a higher resolution as compared to integer numbers. Fixed-point numbers follow *integer.fraction* format and are given by Eq. 1. The total number of digits in a fixed-point number are $\#integer_digits + \#fraction_digits + 1$. An example of a fixed-point number is 010.010 represents 2.5 in decimal (Fig. 4).

Fig. 4 Fixed-point format

Arithmetic operations in fixed-point numbers are similar to that of integers. To convert a fixed-point number of one type with scaling factor K to another type with scaling factor M, the number must be multiplied by $\frac{K}{M}$. If the converted number does not fit the precision digits supported by the underlying system, rounding is to be performed at the cost of accuracy.

To add or subtract on two fixed-point numbers of the same type, it is sufficient to add or subtract the underlying integers. The result can be represented in the same type in the absence of overflow. In case of overflow, it is advisable to sacrifice the fractional bits over integer bits to avoid radical errors.

To multiply two fixed-point numbers, the underlying integers can be multiplied, and the scaling factor of the result is the product of their scaling factors. Similarly, to divide two fixed-point numbers, the underlying integers can be divided, and the scaling factor of the result is a quotient of the scaling factors of the numbers.

The hardware cost of fixed-point arithmetic operations has a little hardware cost over that of integer implementations (Guntoro et al. 2020). The fixed-point arithmetic is prevalent in digital signal processing hardware.

Floating-Point Arithmetic

Out of all the arithmetic formats, the floating-point formats are the most challenging for hardware optimizations. A floating-point number is given by $significand \times base^{exponent}$. The term "floating-point" refers to the fact that the radix-point position is not fixed in the representation and depends on the value of the exponent. There have been a variety of proposals in the literature for the implementations of the floating-point formats. The most prominent is IEEE 754 format shown in Fig. 5 that captures scale factor in the exponent field. All the formats are briefly covered in this exposition.

IEEE 754

To overcome the portability and reproducibility issues of the arbitrarily chosen formats, the IEEE 754 standard was established in 1985 (IEEE Standard for Binary Floating-Point Arithmetic 1985). The IEEE 754-1985 standard was superseded by IEEE 858-1987 (IEEE Standard for Radix-Independent Floating-Point Arithmetic 1987) followed by IEEE 754-2008 (IEEE Standard for Floating-Point Arithmetic 2008). A revision to the IEEE 754-2008 standard was recently released as IEEE 754-2019 (IEEE Standard for Floating-Point Arithmetic 2019). A typical IEEE 754

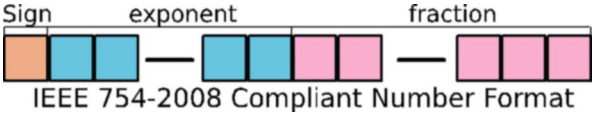


Fig. 5 IEEE 754-2008 format

Table 1 Formats defined by the IEEE 754 standard

Format name	base	Configuration	Decimal digits	Remarks
binay16	2	(5,11)	3.31	–
binary32	2	(8,24)	7.22	basic
binary64	2	(11,53)	15.95	basic
binary128	2	(15,113)	34.02	basic
binary256	2	(19,237)	71.34	–
decimal32	10	(7.58,7)	7	–
decimal64	10	(9.58,16)	16	basic
decimal128	10	(13.58,34)	34	basic

compliant number has three fields: sign, exponent, and fraction (see Fig. 5). Bias is added to the exponent to represent very small and very large quantities. The value of the bias depends on the size of the format.

The IEEE 754 standard defined five basic formats and interchange formats. Table 1 summarized the formats covered in IEEE 754 standard. The decimal digits are calculated by ($\#significand_digits \times \log_{10} base$).

It has been nearly impossible to develop hardware arithmetic units that are fully compliant to IEEE 754 standard (Nandy et al. 2010; Merchant et al. 2016). This is mainly due to the complexities involved in the standard. The standard defines a few special cases such as subnormal numbers, infinity, which makes the full hardware implementations expensive, and hence, some of the functionality is implemented in software. This approach results in performance energy penalties.

Subnormal Numbers

Formerly known as *denormal numbers*, the *subnormal numbers* are used to represent the numbers that are not representable the minimum exponent size (exp_{min}). The numbers that have less than exp_{min} are then represented by shifting the mantissa to the right. Also, the representation uses implicit zero instead of one. The phenomena of approaching zero by right shifting the fractional part are referred to as *gradual underflow*.

There are several performance-related issues in incorporating subnormal numbers. As per the study presented in Schwarz et al. (2003), the performance of subnormal arithmetic when implemented entirely in hardware is comparable to that of normal floating-point performance. This is due to hardware techniques employed for the performance improvement. When implemented in software, the performance of the arithmetic on subnormal numbers is significantly slower compared to the performance of normal floating-point arithmetic. Researchers have also identified that the slower speed can also create a timing channel and possible security leak (Andrysco et al. 2015).

Exceptions

The IEEE 754 standard defines five exceptions. The exception flag is raised when the exception occurs. Handling of the exceptions has been application-specific.

1. Invalid exception: Mathematically undefined operations by default raise an exception
2. Division by zero: Division by zero results in $\pm\infty$ and raises an exception.
3. Overflow: Overflow exception occurs when the result of the computations is beyond the representation strengths of the precision.
4. Underflow: For a very small value that cannot be represented by the underlying precision, result in underflow exception. The distance between the smallest positive normal floating-point value and the smallest negative normal floating-point value is referred to as *underflow gap*. The IEEE 754 standard defined *subnormal* numbers that result in a gradual underflow.
5. Inexact: If the rounded result of a valid arithmetic operation is different from the infinitely precise result, then *inexact* exception occurs.

The division by zero exception is extended to incorporate other operations as well. The decimal floating-point has two more exceptions, *clamped* and *rounded*. The clamp exception is signaled if the exponent of the result is altered to fit the representation. The rounded exception is signaled if the result is rounded in operation.

Not a Number-NaN and Infinity

NaN refers to the data type or value that is undefined or nonrepresentable. Several operations generate NaN. For example, $\frac{\pm 0}{\pm 0}$, $\frac{\pm\infty}{\pm\infty}$, divide by zero or divide by infinity, addition or subtraction of infinity, square root of a negative number, logarithm of a negative number, inverse sign or inverse cosine of a number that is not in the range of [-1,1], and most operations with at least one NaN operand result in NaN. There are two types of NaN defined by the IEEE 754 standard: (i) quiet NaN and (ii) signaling NaN.

Quiet NaN

Quiet NaNs (qNaNs) propagate through the operations without generating any exceptions. However, certain operations, such as format conversions and comparison operations that cannot be performed on NaNs, generate exceptions.

Signaling NaN

Signaling NaNs (sNaNs) generates an exception and then quieted in the process if appropriate. Handling sNaNs is a complex procedure.

In general, there have been several ambiguities in the handling of NaNs in a program. Also, there has been confusion in the handling of qNaNs and sNaNs. In the IEEE 754 standard, NaNs are encoded by all ones in the exponent bits and some nonzero numbers in the fractional part, irrespective of the sign bit. If all the exponent bits are ones and the fractional bits are zero, the bit pattern is regarded as *infinity*. Infinity arises in arithmetic operations.

Rounding Modes

The IEEE 754 standard defines five rounding modes.

1. Round to nearest, ties to even: In this mode, the number is rounded to the nearest number. If the number falls exactly between the two numbers, the number is rounded to the nearest value with an even least significant digit.
2. Round to nearest, ties away from zero: In this mode, the number is rounded to the nearest number. If the number falls midway, it is rounded up in case of a positive number and rounded down in case of a negative number.
3. Round to zero: In this mode, the number is truncated.
4. Round to $+\infty$: In this mode, the number is rounded up.
5. Round to $-\infty$: In this mode, the number is rounded down.

Several implementations support *round to near even, ties to even* rounding mode. If the numerical accuracy is not of paramount importance, then *truncation* is preferred.

A hardware–software codesign-based approach is preferred for IEEE 754 compliant floating-point arithmetic units due to the complexities involved in supporting the full standard (Muller et al. 2018). A few vendors provide compiler or program switches to turn off a subset of the IEEE 754 standard features, resulting in increased performance of the software being executed.

Floating-Point Approximate Circuits

Due to high complexity and limitations in the floating-point arithmetic hardware, computer architects diverged to more optimized circuits by compromising accuracy. The phenomenon is referred to as "approximate computing." Approximate computing also refers to approximation in storage and software-level approximation. The concept of approximation at different levels of abstractions developed due to the applications' requirements for high performance while sustaining the loss of accuracy and numerical precision (Liu et al. 2020).

The rise approximate computing paradigm is also referred to as a "U-turn" concerning the preposition of IEEE 754 floating-point standard, as the standard imposes strict requirements on the formats and numerical correctness in implementations of the basic arithmetic operations (Chaurasiya et al. 2018; IEEE Standard for Floating-Point Arithmetic 2019). The compliance to the IEEE 754 standard is rendered impractical in several machine learning applications where the precision and numerical accuracy are not of paramount importance while the energy efficiency and run-time are (Saxena et al. 2021).

Historically, it is not possible to backtrack the developments in the field of approximate computing as the field might be as old as the age of the first-ever developed computing device. Once the digital systems are devised, the approximation is introduced due to the quantized digital form. However, the renaissance transpired post-breakdown of Dennard scaling. In the era where technology scaling might not be possible anymore, approximate computing might provide the required mettle to meet power and performance requirements.

Posit Arithmetic

Posit arithmetic is proposed as a drop-in replacement for the IEEE 754 compliant arithmetic. It is claimed that the posit arithmetic format offers several absolute advantages over its IEEE 754 compliant counterpart of the same size. An m-bit posit number packs more information per bit compared to its IEEE 754 compliant single-precision number where m can be any of the bit-width format supported by the IEEE 754 standard. The posit format has only one exception called Not-a-Real (NaR) (Fig. 6).

Since its inception in 2017 by John Gustafson (Gustafson and Yonemoto 2017), there have been several studies on posit. Broadly, these studies can be classified as hardware-based investigations and software-based analyses. In the hardware-based implementations, there have been general-purpose parametric designs (Chaurasiya et al. 2018) and application-specific designs (Nambi et al. 2020). The software and application analyses are carried-out using either *SoftPosit* (Cerlane Leong 2018) or *Universal* (Omzigt et al. 2018). A preliminary study of posits vis-à-vis floats suggests that the numbers represented in posit format are more tolerant to single and double bit-flips compared to their IEEE 754 compliant counterparts (Alouani et al. 2021). In the case of bit-flips, the probability that a number will result in NaN is much higher in floats compared to posits.

To facilitate hardware–software-arithmetic codesign, there have been a few posit hardware implementations integrated into a RISC-V core. A qualitative comparison of the implementations is shown in Table 2. The only implementation that supports

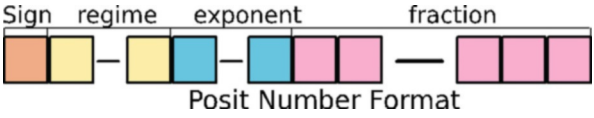


Fig. 6 Posit format for non-exception values

Table 2 RISC-V-based posit arithmetic implementations in the literature

Impl.	Parametric	Application study	RISC-V integration	Posit custom instruction support	Open source	SoftPosit porting	Quire support
PERI (Tiwari et al. 2019)	✓	✗	✓	✗	✗	✗	✗
PERC (ThoughtWorks 2020)	✓	✗	✓	✗	✓	✗	✗
CRISP[a]	✗	✗	✓	✗	✗	✗	✗
Saxena (Saxena et al. 2021)	✓	✓	✓	✗	✗	✗	✗
Clarinet (Sharma et al. 2023)	✓	✓	✓	✓	✓	✓	✓

[a]Based on publicly available data (Calligo Technologies 2020)

quire is *Clarinet* and posit custom instructions. The rest of the implementations leverage the floating-point instructions for the posit hardware utilization.

One feature of posit is support for *quire*. The quire accumulator is a functionality that can be implemented in software or hardware. The software libraries such as SoftPosit (Cerlane Leong 2018) and Universal (Omzigt et al. 2018) support quire in software, while Clarinet presented in Sharma et al. (2023) supports quire register as a hardware feature. The format of quire register is shown in Fig. 7. The format of the quire register is similar to the fixed-point format with carry-guard as an extra field.

Other Formats

Since the inception of floating-point arithmetic, there have been a variety of number format proposals in the literature. Here, some of them are covered.

BF16

The bfloat16 (BF16) format is the truncated version of IEEE 754 single-precision floating-point format (see Fig. 8). The BF16 format was particularly designed to accelerate machine learning, especially training, and near-sensor computing (Tagliavini et al. 2018).

The format retains 8 exponent bits of IEEE 754 single-precision format, while it supports only 8-bit precision (7 fractional bits), resulting in reduced accuracy. The BF16 format has a higher dynamic range and lower precision compared to half-precision format (binary6). BF16 numbers are not usable for integer calculations.

Similar to IEEE 754 compliant formats, a lot of binary bit patterns (256) are wasted in defining the NaN formats. Also, the BF16 format defines 2-bit patterns for $+\infty$ and $-\infty$ and 2-bit patterns for $+0$ and -0. The precision of BF16 is between two and three decimal digits.

Several vendors have adopted BF16 for their platforms. For example, Intel has adopted it for its Nervana and FPGAs (Intel Unveils Nervana Neural Net L-1000 for accelerated AI training 2020). Other major vendors such as Google, ARM, and AMD have also released products based on BF16 format.

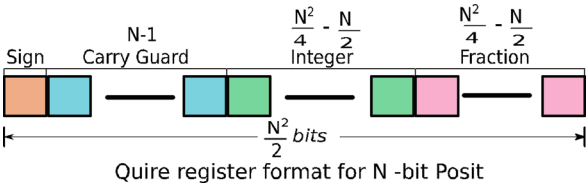


Fig. 7 Quire format for N-bit posit

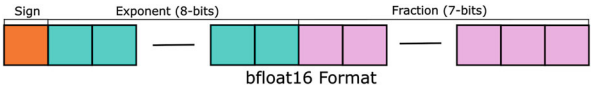


Fig. 8 bfloat16 (BF16) format

TensorFlow-32

TensorFlow-32 (TF32) is the math mode introduced in NVidia A100 GPUs. The core runs TF32 arithmetic that has 10 bits in fractional part and 8 bits in exponent (HPC up to 20x TensorFloat-32 in the A100 GPU Accelerates AI Training 2020). The aim is to strike a balance to achieve performance and required accuracy for machine learning training.

There have been other formats such as Microsoft Binary Format (MBF), mini-float, and IBM Hexadecimal Floating Point (HFP) in the literature. However, with the standardization of floating-point format, the IEEE 754 standard implementations were preferred over other formats, except for machine learning applications, where customized formats are more popular.

Hardware Implementations

Design and development of arithmetic circuits is driven by the target application. Especially, the data type to be supported at high-level programs has direct reflection on the arithmetic to be implemented in hardware. Typically, the instruction set architecture (ISA) dictates the underneath implementation though the implementation details are invisible to the programmer or compiler designer.

Adders

Binary half-adders and full-adders form a fundamental component in the larger adder circuits that are prevalent today. The truth tables and gate-level designs are depicted in Fig. 9. The binary half-adder can be implemented using one XOR gate and one AND logic gate. The binary half-adder can be expressed as $S = A$ XOR B and $C = A$ AND B, where S is the summation of inputs A and B, and C is the carry bit. The gate-level circuit and the truth table of a half-adder are shown in Fig. 9 (left). The binary full-adder can be implemented using two XOR, two AND, and one OR gates as shown in Fig. 9. The summation in a binary full-adder is expressed as $S = A$ XOR B XOR C_{in}, and the output carry is expressed as $C_{OUT} = A$ AND B OR (C_{in} AND (A XOR B)).

Ripple-Carry Adder

To add two N-bit numbers, multiple full-adders can be cascaded where carry ripples through these adders (see Fig. 10). C_{in} of each adder is C_{out} of the previous adder. The first full-adder can be replaced by a half-adder in a ripple-carry adder assuming that C_{in} for the N-bit adder is 0. Ripple-carry adder is generally slow since each full-adder requires carry input to process the operation. This forms a critical path in the adder. Furthermore, each full-adder requires three levels of logic (see Fig. 9 (right)).

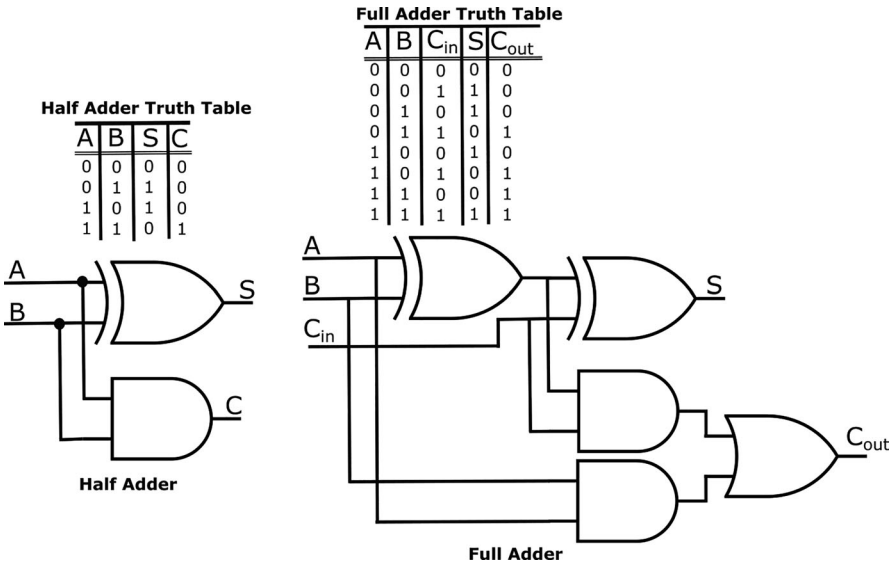


Fig. 9 Binary half- (left) and full-adders (right) and their respective truth tables

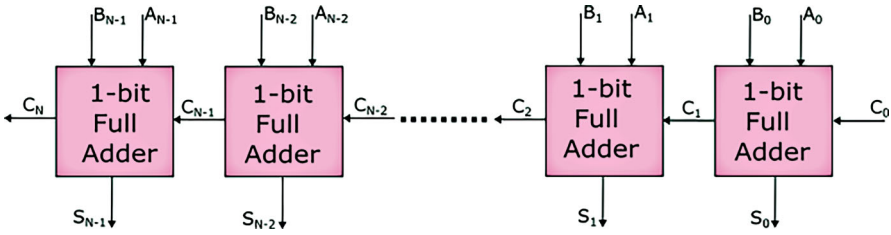


Fig. 10 Ripple-carry adder

Carry-Lookahead Adder

A carry-lookahead adder (CLA) overcomes the performance issue in the ripple-carry adder by processing the carry bit faster. The CLA computes one or more carry bits before the sum, which reduces the wait time for the carry bits, resulting in faster addition. Konrad Zuse implemented the first CLA (Rojas 2014). Kogge–Stone adder (KSA) and Brent–Kung adder (BKA) are the types of CLA (Fig. 11).

Multipliers

Multiplier plays a key role in an arithmetic logic unit (ALU) of a central processing unit (CPU). The early implementations of multipliers were software-based, using adder circuits; however, with the advent of hardware–software codesign techniques, more sophisticated hardware blocks were established. This resulted in

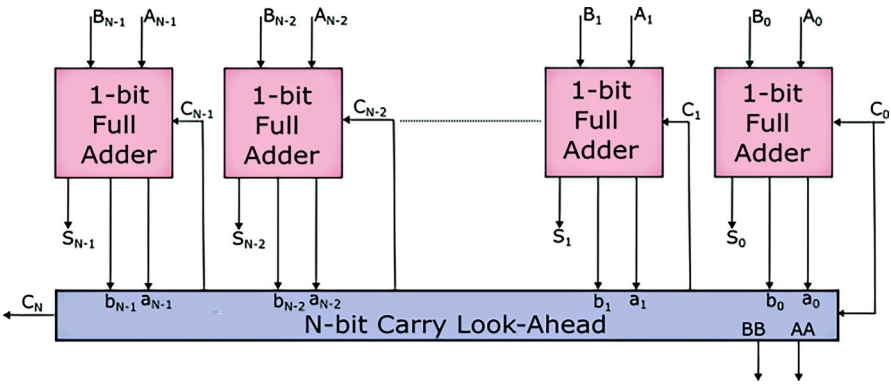


Fig. 11 Carry-lookahead adder

energy-efficient implementations of multipliers. Furthermore, several techniques, such as Booth's multiplication and Wallace tree, were invented to improve the hardware efficiency.

Dividers

Scientific applications heavily rely on division operations. In the early days, dividers were implemented in software due to the complex hardware requirements. Furthermore, divider circuits were extremely expensive in terms of area and energy (Obermann and Flynn 1997). However, later implementations transitioned to hardware due to improvements in energy efficiency and performance (Obermann and Flynn 1997). Over the years, integer, fixed-point, and floating-point dividers have been implemented in ASICs and FPGAs (Hemmert and Underwood 2007; Ugurdag et al. 2017). Even modern GPGPUs and multicore CPUs provide extensive support for division operations in hardware.

Square Root

Similar to dividers, square root operations are expensive in hardware (Mopuri and Acharyya 2017). Additionally, there are fewer applications that require square root operations, and in these applications, the occurrences of square roots are less frequent. Due to this, square root operations are typically implemented in software. However, in applications where energy is a major constraint, a hardware-implemented square root is preferred (Hasnat et al. 2017).

Conclusion

Computer arithmetic formats and implementations have been an active area of research for decades due to new application domains. Historically, there have

been several proposals in the literature for different formats, and standardization efforts have been made to ensure uniformity and reproducibility. Especially for the floating-point arithmetic, the efforts have been significant due to the floating-point architectures' hardware complexities. Later, the computer architects diverged to more optimized arithmetic formats suitable for the applications. Several new formats were invented, such as TF32, posit, and bfloat16, that are envisioned for replacement of IEEE 754 compliant formats. Hardware design for these new formats has been challenging since the applications areas such as edge computing require extremely small area and energy footprints. Also, with the advancements in the technologies, with the arrival of post-CMOS technologies in the offing, there are several research opportunities in identifying the right format that fits the novel technologies.

References

Alouani I, Ben Khalifa A, Merchant F, Leupers R (2021) An investigation on inherent robustness of posit data representation. In: Proceedings of the international conference on vlsi design (VLSID)

Andrysco M, Kohlbrenner D, Mowery K, Jhala R, Lerner S, Shacham H (2015) On subnormal floating point and abnormal timing. In: 2015 IEEE symposium on security and privacy, pp 623–639

Calligo Technologies (2020) Posit Numeric Unit (PNU-IP). https://calligotech.com/posit-numeric-unit-pnu-ip/. Accessed 17 Dec 2020

Cerlane Leong (2018) Softposit version 0.4.1rc

Chaurasiya R, Gustafson J, Shrestha R, Neudorfer J, Nambiar S, Niyogi K, Merchant F, Leupers R (2018) Parameterized posit arithmetic hardware generator. In: 2018 IEEE 36th International conference on computer design (ICCD), pp 334–341

Garner HL (1976) A survey of some recent contributions to computer arithmetic. IEEE Trans Comput C-25(12):1277–1282

Goldberg D (1991) What every computer scientist should know about floating-point arithmetic. ACM Comput Surv 23(1):5–48 (1991)

Guntoro A, De La Parra C, Merchant F, De Dinechin F, Gustafson JL, Langhammer M, Leupers R, Nambiar S (2020) Next generation arithmetic for edge computing. In: 2020 Design, automation test in Europe conference exhibition (DATE), pp 1357–1365

Gustafson JL, Yonemoto I (2017) Beating floating point at its own game: posit arithmetic. Supercomput Front Innov Int J 4(2):71–86

Hasnat A, Bhattacharyya T, Dey A, Halder S, Bhattacharjee D (2017) A fast FPGA based architecture for computation of square root and inverse square root. In: 2017 Devices for integrated circuit (DevIC), pp 383–387

Hemmert KS, Underwood KD (2007) Floating-point divider design for FPGAs. IEEE Trans Very Large Scale Integr VLSI Syst 15(1):115–118

HPC up to 20x TensorFloat-32 in the A100 GPU Accelerates AI Training (2020) https://blogs.nvidia.com/blog/2020/05/14/tensorfloat-32-precision-format/. Accessed 17 Dec 2020

IEEE Standard for Binary Floating-Point Arithmetic (1985) ANSI/IEEE Std 754-1985, pp 1–20

IEEE Standard for Floating-Point Arithmetic (2008) IEEE Std 754-2008, pp 1–70

IEEE Standard for Floating-Point Arithmetic (2019) IEEE Std 754-2019 (Revision of IEEE 754-2008), pp 1–84

IEEE Standard for Radix-Independent Floating-Point Arithmetic (1987) ANSI/IEEE Std 854-1987, pp 1–19

Intel (2021) 8087 Math Coprocessor. http://pdf.datasheetcatalog.com/datasheets/2300/45014_DS.pdf. Accessed 22 Jan 2021

Intel Unveils Nervana Neural Net L-1000 for Accelerated AI Training (2020) https://venturebeat.com/2018/05/23/intel-unveils-nervana-neural-net-l-1000-for-accelerated-ai-training/. Accessed 17 Dec 2020

Ken Shirriff's Blog (2021) 8087 Math Coprocessor Die. http://www.righto.com/2020/05/extracting-rom-constants-from-8087-math.html. Accessed 22 Jan 2021.

Liu W, Lombardi F, Shulte M (2020) A retrospective and prospective view of approximate computing [point of view. Proc IEEE 108(3):394–399

Marasa JD, Matula DW (1973) A simulative study of correlated error propagation in various finite-precision arithmetics. IEEE Trans Comput C-22(6):587–597

Merchant F, Choudhary N, Nandy SK, Narayan R (2016) Efficient realization of table look-up based double precision floating point arithmetic. In: 2016 29th International conference on VLSI design and 2016 15th International conference on embedded systems (VLSID), pp 415–420

Mopuri S, Acharyya A (2017) Low-complexity methodology for complex square-root computation. IEEE Trans Very Large Scale Integr VLSI Syst 25(11):3255–3259

Muller J-M, Brunie N, de Dinechin F, Jeannerod C-P, Joldes M, Lefèvre V, Melquiond G, Revol N, Torres S (2018) Handbook of floating-point arithmetic, 2nd edn. Springer

Nambi S, Ullah S, Lohana A, Satyendra Sahoo S, Merchant F, Kumar A (2020) Expan(n)d: Exploring posits for efficient artificial neural network design in FPGA-based systems

Nandy S, Balakrishnan S, Merchant F, Baluni A (2010) A fully pipelined modular multiple precision floating point multiplier with vector support. In: 2010 International symposium on electronic system design, Los Alamitos, Dec 2011. IEEE Computer Society, pp 45–50

Obermann SF, Flynn MJ (1997) Division algorithms and implementations. IEEE Trans Comput 46(8):833–854

Omzigt T (2018) Universal: a header-only C++ template library for universal number arithmetic

Palmer JF (1980) The Intel 8087 numeric data processor. In: International workshop on managing requirements knowledge, Los Alamitos. IEEE Computer Society, p 887

Rojas R (2014) The Z1: architecture and algorithms of Konrad Zuse's first computer. CoRR, abs/1406.1886

Saxena V, Merchant F, Reddy A, Gustafson JL, Jonathan N, Sangeeth N, Leupers R (2021) Brightening the optical flow through posit arithmetic. In: International symposium on quality electronic design (ISQED)

Schwarz EM, Schmookler M, Trong SD (2003) Hardware implementations of denormalized numbers. In: Proceedings 2003 16th IEEE symposium on computer arithmetic, pp 70–78

Sharma NN, Jain R, Pokkuluri MM, Patkar SB, Leupers R, Nikhil RS, Merchant F (2023) Clarinet: a quire-enabled RISC-V-based framework for posit arithmetic empiricism. J Syst Archit 135:102801

Swartzlander EE, Alexopoulos AG (1975) The sign/logarithm number system. IEEE Trans Comput C-24(12):1238–1242

Tagliavini G, Mach S, Rossi D, Marongiu A, Benin L (2018) A transprecision floating-point platform for ultra-low power computing. In: 2018 Design, automation test in Europe conference exhibition (DATE), pp 1051–1056

ThoughtWorks (2020) Posit Enhanced Rocket Chip (PERC). https://www.thoughtworks.com/engineering-research/perc. Accessed 17 Dec 2020

Tiwari S, Gala N, Rebeiro C, Kamakoti V (2019) PERI: a posit enabled RISC-V core, pp 1–14

Ugurdag HF, de Dinechin F, Gener YS, Gören S, Didier L-S (2017) Hardware division by small integer constants. IEEE Trans Comput 66(12):2097–2110

Architectures for Scientific Computing

12

Farhad Merchant

Contents

Introduction . . . 402
Definitions . . . 403
  Scientific Computing . . . 403
  Multicore Architectures . . . 404
  Manycore Architectures . . . 405
  Field-Programmable Gate Arrays . . . 405
  Coarse-Grained Reconfigurable Architectures . . . 405
  Custom Architectures . . . 407
Multicore Architectures . . . 407
General Purpose Graphics Processing Units . . . 408
Field-Programmable Gate Arrays . . . 410
Coarse-Grained Reconfigurable Architectures . . . 410
Conclusion . . . 412
References . . . 412

Abstract

Scientific computing workloads are compute-intensive. Especially, the dense linear algebra (DLA) computations are embarrassingly parallel in nature. Besides, these computations exhibit a high computation-to-communication ratio if written in terms of Level-3 basic linear algebra subprograms (BLASs). On the other hand, if not optimized appropriately, the DLA computations can incur significantly higher run-time than their optimized counterparts. Several architectures are explored to accelerate these computations. Some of the examples are multicore, manycore, field-programmable gate arrays, and coarse-grained

F. Merchant (✉)
University of Groningen, Groningen, The Netherlands
e-mail: f.a.merchant@rug.nl

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_16

reconfigurable architectures. For off-the-shelf platforms such as multicore and manycore architectures, a library-based approach is adopted where highly optimized BLAS is developed. This optimized software is then further used to develop more complex algorithms, such as LU and QR factorization, that belong to the linear algebra package (LAPACK). For multicore and manycore architectures, appropriate programming methodology is used to schedule BLAS and LAPACK kernels on these architectures. For reconfigurable architectures such as field-programmable gate arrays and coarse-grained reconfigurable architectures, hardware–software codesign is carried out to arrive at optimal power and performance for scientific computing workloads. This chapter discusses some of the software and hardware–software codesign techniques.

Keywords

Multicore · Manycore · CGRA · FPGA

Introduction

Scientific computing (also known as computational science) has many large-scale problems that require significant amount of run-time (Bates et al. 2005). Furthermore, the scientific computing workloads are compute and memory-intensive, consuming megawatts of energy. Scaling the algorithms and architectures has become challenging. Several hardware platforms are adopted to perform scientific computing. These platforms are multicore (Dongarra and Luszczek 2011), manycore (Krüger and Westermann 2003), field-programmable gate arrays (FPGAs) (Goetting et al. 1995), and coarse-grained reconfigurable architectures (CGRAs) (Das et al. 2014). Out of these platforms, the adoption of multicore and manycore has been rapid as these platforms require a library-based approach. A highly optimized library is developed, which is then connected to the high-level language through application programming interfaces (APIs) (Dongarra and Luszczek 2011; Bolz et al. 2003; Merchant et al. 2018). On the other hand, hardware–software codesign-based approaches are adopted for FPGAs and CGRAs. In this approach, hardware customizations are carried out to efficiently execute the target scientific computing code. One such example is a dot-product unit that can perform dot-product of two input vectors in hardware (Merchant et al. 2015; Vreca et al. 2020). Furthermore, in the case of CGRAs, compute-intensive code regions are identified to design custom instructions for their execution in hardware (Das et al. 2014). This minimizes memory access, resulting in lower energy at the cost of more hardware resources.

This chapter covers the basic concepts related to various state-of-the-art architectures and discuss their applicability and potential for scientific computing. This chapter is architecture-centric, with a flavor of programming methods for architecture. The major contributions in this chapter are as follows:

- Library-based approaches for the implementation of scientific computing workloads on multicore and manycore architectures
- Hardware–software codesign approaches for the implementation of scientific computing workloads on FPGA and CGRAs

The rest of the chapter is organized as follows: In section "Definitions," several definitions are covered to help the readers. In section "Multicore Architectures," scientific computing on multicore is discussed. In section "General Purpose Graphics Processing Units," libraries developed for scientific computing on GPUs are discussed. In sections "Field-Programmable Gate Arrays" and "Coarse-Grained Reconfigurable Architectures," the hardware–software codesign for FPGAs and CGRAs is discussed. The chapter is concluded in section "Conclusion."

Definitions

To understand the concepts better, a few fundamental definitions are delved into in the following section.

Scientific Computing

Scientific computing, also known as computational science or scientific computation, is an art (or science) that uses computing to solve complex physical problems. The target physical problems are converted to mathematical formulations. These mathematical formulations are then executed on a computing infrastructure using application peripheral interfaces (APIs) (Higham 1993). The field of scientific computing involves complex mathematical models, APIs that can be written in terms of a program to execute these models, and architectures that can execute these programs. There are multiple challenges in the domain of scientific computing. The first is ease of programming, where the overarching goal is to ensure that the target architecture is programmable with minimum efforts where the efforts are not quantified. The second goal is to extract the performance of the target architecture. The third and final approach is to perform hardware–software codesign, where the algorithms are redesigned to exploit the target architectures, and the architectures are customized to execute the algorithms efficiently. A recommended way to enhance the performance of scientific computing programs is to rewrite the code as one or multiple dwarfs (Asanovic et al. 2009). The authors recommend further reading (Asanovic et al. 2009). This chapter discusses dense linear algebra (DLA) compute architectures since DLA is found in many applications of scientific computing, such as computational fluid dynamics to machine learning (Bates et al. 2005).

Multicore Architectures

A multicore processor is a computing platform that consists of multiple (two or more) processing units or elements on a single integrated circuit. Post-breakdown of Dennard scaling, the multicore processor gained popularity as they allowed the performance improvement through packing as many processor cores as possible on a single chip (Bohr 2007) (see Fig. 1). Multicore architectures have been significantly used for scientific computing where symmetric multiprocessing (SMP) is used to achieve the desired performance (see Fig. 2). SMP is achieved by connecting two or more identical processors to a shared main memory. The other resources, such as the system bus, input/output devices, and operating systems, are shared among these processors. Due to the complexity of cores, cache coherency restricts the scaling of multicore processors.

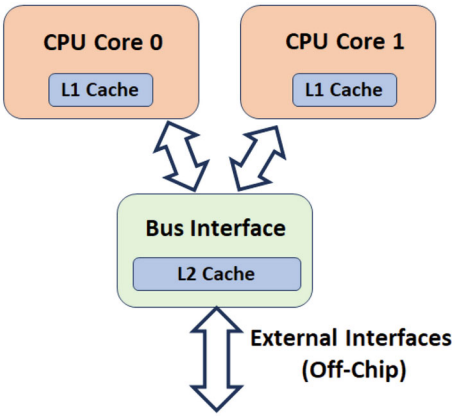


Fig. 1 Multicore architecture

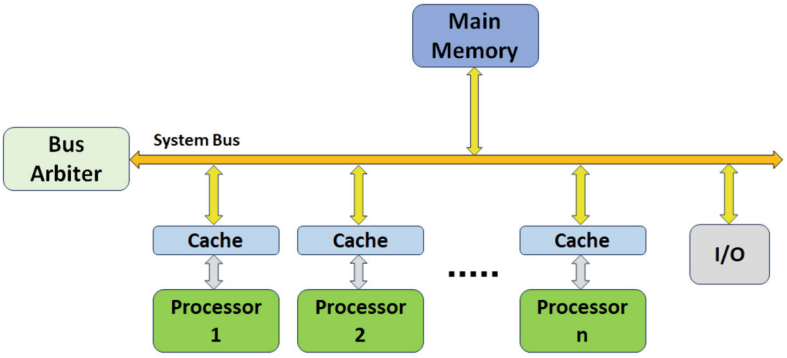


Fig. 2 Symmetric multiprocessing system

Manycore Architectures

Manycore processors are a class of multicore processors where the processor cores are simplified. The cores are mostly small, with minimal control circuits and small local memories. A host computer is required to schedule computations on manycore architecture. The manycore processor architectures offer a higher degree of explicit parallelism (see Fig. 3). The scientific computing code then exploits this parallelism. Manycore processors do not have concepts such as message passing, direct memory access, partitioned global address spaces, and noncoherent caches. GPUs are one example of manycore processors.

Field-Programmable Gate Arrays

The CMOS field-programmable gate arrays (FPGAs) are seen as sea-of-gates that have logic gates and programmable interconnect (Goetting et al. 1995) (see Fig. 4). The FPGAs are usually programmed using hardware description languages. However, in the last decade, there has been significant progress in the design and development of high-level synthesis tools that allow configuring FPGAs using languages such as C. FPGAs exhibit fine-grained bit-level parallelism that can be exploited in many scientific applications such as graph mining and machine learning (Jaiyeoba et al. 2023; Dai et al. 2017; Nechi et al. 2023).

Coarse-Grained Reconfigurable Architectures

Coarse-grained reconfigurable architectures (CGRAs) are architectures that have an array of function units connected through an on-chip network (see Fig. 5). The network could be circuit-switched or packet-switched (Cong et al. 2014; Nimmy

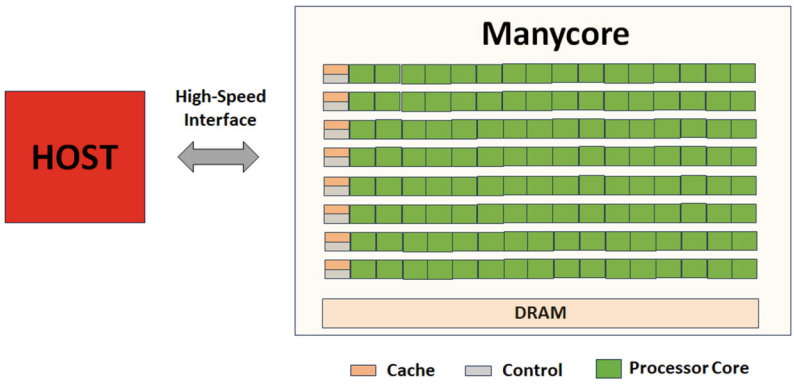


Fig. 3 Manycore architecture

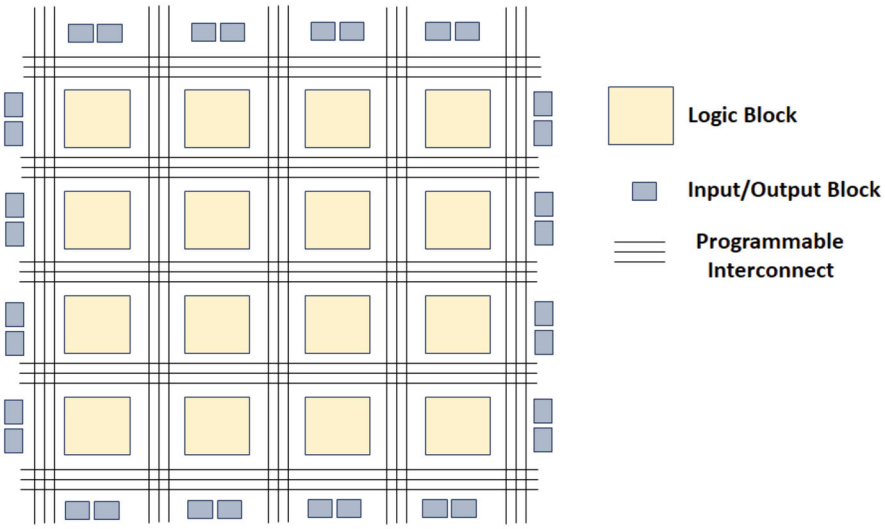


Fig. 4 A generic FPGA architecture

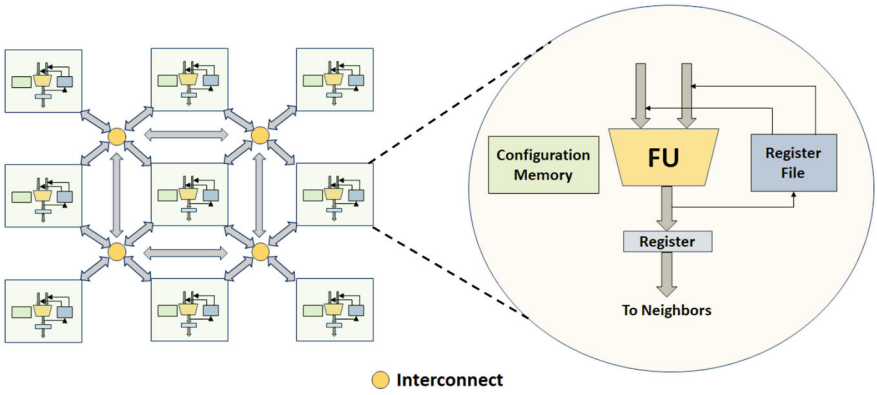


Fig. 5 A generic coarse-grained reconfigurable architecture with function units and network-on-chip

et al. 2008). The granularity of operation in a CGRA is defined by the granularity of operations in these function units (Das et al. 2014; Anderson et al. 2021). CGRAs are well suited as systolic scheduling, which is, in turn, ideal for matrix operations (Merchant et al. 2014; Rákossy et al. 2014b; Mahadurkar et al. 2014; Rákossy et al. 2014a).

Custom Architectures

Custom integrated circuits are less preferred as a computing substrate for scientific computing due to the lack of flexibility. However, tailored designs, together with reprogrammable /reconfigurable fabric, are used to accelerate the target scientific computing workloads. One such example is CGRAs.

Multicore Architectures

As described earlier, multicore architectures have relatively complex cores. The following challenges are involved while executing scientific computing code on a multicore architecture:

- Sustaining computation to communication ratio where communication is defined as the movement of data among the different levels of memory
- Programmability of the architecture

For the first challenge, a library-based approach is adopted where highly optimized scientific libraries are developed. These library routines are capable of minimizing the pipeline stalls in the code. To use these optimized libraries, application programming interfaces (APIs) are developed and then used to write scientific application code. These APIs address the second challenge, as it is not possible for a regular programmer to efficiently program on a multicore architecture to extract the performance out of it.

One such example is parallel linear algebra software for multicore architectures (PLASMA) shown in Fig. 6. PLASMA uses highly optimized basic linear algebra subprograms (BLASs) to improve the efficiency of the target application. PLASMA offers solutions to linear systems of equations, least square problems, eigenvalue problems, and singular value problems (Dongarra and Luszczek 2011). PLASMA supports thread-based routines, and later, it was implemented using OpenMP (Dongarra et al. 2019).

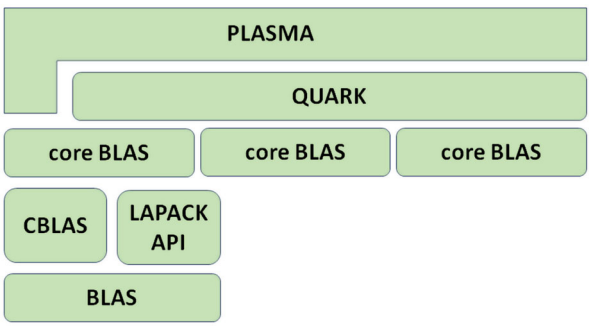


Fig. 6 PLASMA software stack

General Purpose Graphics Processing Units

Graphics processing units (GPUs) are a class of manycore architectures. GPUs were originally developed for graphics processing, especially shaders, while applications were executed on a central processing unit (also known as a host machine). The advent of GPUs dates back to the 1970s–1980s when the target application was video games. Originally, shading languages were used to program GPUs. In a typical CPU-GPU architecture, GPUs were external, connected to the CPU through a bus such as PCI Express (see Fig. 7). Initial GPUs did not support floating-point arithmetic.

In early 2001, general-purpose computing became popular on GPUs. This was due to the improved programmable shaders and support for floating-point arithmetic on GPUs. It was observed that the GPUs are suitable for applications that involve matrix computations (Krüger and Westermann 2003; Bolz et al. 2003). Historically, GPUs were programmed using OpenGL or DirectX. Both these provide application programming interfaces to write application code. However, both application programming interfaces are targeted for graphics, and hence programming for general purpose computing became cumbersome with these interfaces as it required the knowledge of graphical concepts. Later, the advent of computer-unified device architecture (CUDA) allowed programmers to ignore the graphical concepts. Thus, a programmer could think of high-performance computing concepts while programming GPUs. In the following, some of the concepts of GPUs that assist in scientific computing are described.

Arithmetic format support GPUs targeted for graphics processing mostly supported integer arithmetic ranging from 8 bits to 32 bits. However, repurposing GPUs for scientific computing required them to support the floating-point format as most of the scientific applications required 32 or 64-bit floating-point computations. Incorporating high-precision floating-point hardware on GPUs gave rise to the trade-off between accuracy and performance, two contrasting goals that are extremely important for scientific computing.

Vectorization Since most GPUs were repurposed for applications that consist of matrix computations, vectorization support became obvious. This gave rise to concepts related to single-instruction multiple data (SIMD) support on GPUs.

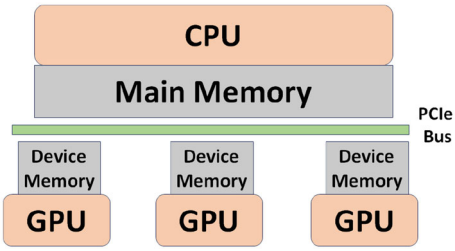


Fig. 7 CPU-GPU (heterogeneous) architecture

Caches Traditionally, GPUs that were used for graphics processing did not require caches. This was mainly due to the GPU processing data being rendered on the display. However, once the GPUs were repurposed for general-purpose computing, the need for caches became apparent as data locality became critical.

Register files Most state-of-the-art GPUs support large register files due to vectorization. Besides, the large register files reduce the context switching latency.

Stream processing In stream processing, given a sequence of data, a series of operations are applied to the data. The sequence of data is known as a stream, while the series of operations is known as a kernel. Considering that most of the matrix computations are vectorized, the support for stream processing on GPUs makes them an ideal choice for these matrix computations (Merchant et al. 2018).

A library-based approach is adopted for the implementation of the DLA. The two prominent software packages are matrix algebra on GPU and multicore architectures (MAGMA) library from the innovative computing laboratory at the University of Tennessee, Knoxville (USA) and CUBLAS, developed by NVIDIA.

The objective of the MAGMA library is to provide the functionality of liner algebra package (LAPACK) (Anderson et al. 1999) and scalable linear algebra package (ScaLAPACK) (Blackford et al. 1997) on heterogeneous architectures (see Fig. 8). MAGMA uses a hybridization methodology where the target algorithm

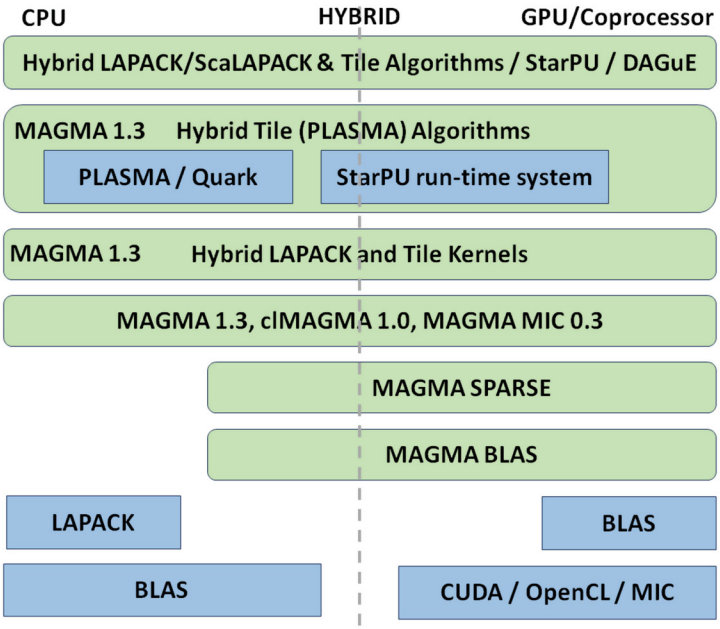


Fig. 8 MAGMA software stack

is divided into tasks of varying granularity. These tasks are then scheduled on the heterogeneous system statically or dynamically. Nonparallelizable tasks are executed on the CPU. The parallelizable tasks that are in the form of Level-2 or Level-3 BLASs are scheduled on GPUs. MAGMA is a high-performance architecture and supports various formats such as single precision (S), double precision (D), single-precision complex numbers (C), and double-precision complex numbers (Z).

CUBLAS is a CUDA-accelerated API developed by NVIDIA that supports Level-1 (vector–vector), Level-2 (matrix–vector), and Level-3 (matrix–matrix) operations. The CUBLAS library is highly optimized for NVIDIA GPUs.

Field-Programmable Gate Arrays

Classical computing platforms such as multicore and GPUs are capable of exploiting coarse-grained parallelism in scientific computing workloads. FPGAs are capable of exploiting fine-grained and coarse-grained parallelism. Furthermore, FPGAs offer unique advantages compared to CPUs and GPUs (Kestur et al. 2010). These advantages are low latency, high performance, and superior energy efficiency. On the other hand, there are a few challenges involved while implementing a scientific computing application on an FPGA. These challenges are as follows:

- High specification to deployment time: This is mainly due to the amount of time required to program FPGAs using hardware description languages.
- Hardware debugging: Since the designs are implemented in the form of hardware description languages, the debugging becomes tedious. Many FPGA vendors do not have the required support for debugging designs on FPGAs. This results in trial-and-error-based methods, consuming a great amount of time.
- Limited resources: Often, the amount of FPGA resources is not sufficient to implement the entire design on an FPGA. This results in using an FPGA cluster, which impacts the performance or requires a complex data scheduling mechanism to reuse the resources, again having a similar performance bottleneck.

In the recent years, there are C and OpenCL-based front ends provided to program FPGAs apart from hardware description languages-based front ends. This has resulted in a great popularity among FPGA enthusiasts for using FPGAs for high-performance scientific codes (De Matteis et al. 2020).

Coarse-Grained Reconfigurable Architectures

CGRAs offer a unique balance between flexibility and performance. They occupy the middle ground between fixed-function custom integrated circuits and fully programmable/reconfigurable architectures such as multicores or FPGAs. For this reason, CGRAs have become popular for embedded and scientific computing

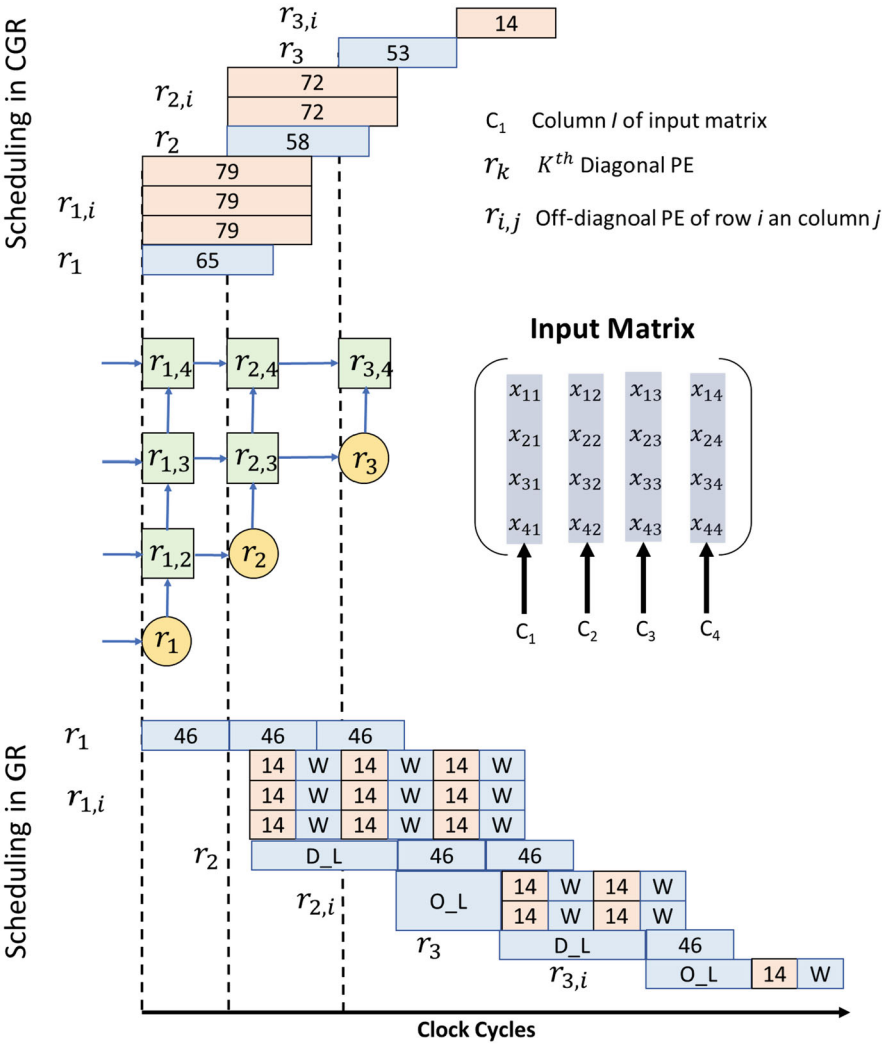


Fig. 9 4×4 matrix scheduling on systolic array for classical and column-wise Givens rotation. (Reproduced from Merchant et al. 2014)

applications. Another major advantage the CGRAs have is their array-like structure, making them highly suitable for matrix computations (Tan et al. 2022) (see Fig. 9).

Especially, CGRAs are amenable to systolic scheduling as suggested by the authors in Merchant et al. (2014), Rákossy et al. (2014b), and Mahadurkar et al. (2014). Despite the promises in CGRAs for scientific computing applications, large-scale deployment remains a challenge. These challenges are as follows:

- High degree of hardware–software codesign: Implementing a scientific application on a CGRA requires the skills of hardware and software designs. This results in very large design times for this kind of architecture.
- Lack of standardization: For CGRAs, there is no standardization for the implementation aspects. This can help extract performance; however, nonstandardization makes the hardware and software non-portable when it comes to the portability of code and architecture components.
- Complex scheduling mechanisms: CGRAs require a host processor to schedule operations on the function units. This results in complex and programmer-aware scheduling mechanisms.
- Poor programmability: Often, CGRAs have extremely poor programmability due to missing high-level APIs.
- Compiler techniques: Compilation of a high-level programming language for CGRAs remains a challenge forever.

Due to the many of the above challenges, the adoption of CGRAs remains limited for scientific applications. On the other hand, many CGRA-like architectures are adopted in embedded computing due to deterministic and constrained workloads.

Conclusion

The implementation of scientific computing workloads on state-of-the-art architectures is challenging due to memory bottlenecks. Several architectural enhancements have been made for multicore and manycore architectures, such as CPUs and GPGPUs, while hardware–software codesign is employed for FPGAs and CGRAs. There is no clear winner among these platforms, as various applications achieve different performance and energy efficiency on these architectures. This is primarily due to the properties of the applications, such as the flow of data and computations. Research targeting high-performance and low-power architectures has significantly improved these aspects. In the future, it will be exciting to use emerging technologies, such as resistive random-access memory-based in-memory computing, to design reconfigurable and reprogrammable architectures to enhance energy efficiency (Staudigl et al. 2022).

References

Anderson E, Bai Z, Bischof C, Blackford S, Demmel J, Dongarra J, Du Croz J, Greenbaum A, Hammarling S, McKenney A, Sorensen D (1999) LAPACK users' guide, 3rd edn. SIAM, Philadelphia

Anderson J, Beidas R, Chacko V, Hsiao H, Ling X, Ragheb O, Wang X, Yu T (2021) CGRA-ME: an open-source framework for CGRA architecture and cad research: (invited paper). In: 2021 IEEE 32nd international conference on application-specific systems, architectures and processors (ASAP), pp 156–162

Asanovic K, Bodik R, Demmel J, Keaveny T, Keutzer K, Kubiatowicz J, Morgan N, Patterson D, Sen K, Wawrzynek J, Wessel D, Yelick K (2009) A view of the parallel computing landscape. Commun ACM 52(10):56–67

Bates PD, Lane SN, Ferguson RI (2005) Computational fluid dynamics: applications in environmental hydraulics. Wiley, New York

Blackford LS, Choi J, Cleary A, D'Azeuedo E, Demmel J, Dhillon I, Hammarling S, Henry G, Petitet A, Stanley K, Walker D, Whaley RC, Dongarra JJ (1997) ScaLAPACK user's guide. Society for Industrial and Applied Mathematics, Philadelphia

Bohr M (2007) A 30 year retrospective on Dennard's MOSFET scaling paper. IEEE Solid-State Circuits Soc Newsl 12(1):11–13

Bolz J, Farmer I, Grinspun E, Schröder P (2003) Sparse matrix solvers on the GPU: conjugate gradients and multigrid. ACM Trans Graph 22(3):917–924

Cong J, Huang H, Ma C, Xiao B, Zhou P (2014) A fully pipelined and dynamically composable architecture of CGRA. In: 2014 IEEE 22nd annual international symposium on field-programmable custom computing machines, pp 9–16

Dai G, Huang T, Chi Y, Xu N, Wang Y, Yang H (2017) ForeGraph: exploring large-scale graph processing on multi-FPGA architecture. In: Proceedings of the 2017 ACM/SIGDA international symposium on field-programmable gate arrays, FPGA'17. Association for Computing Machinery, New York, pp 217–226

Das S, Madhu K, Krishna M, Sivanandan N, Merchant F, Natarajan S, Biswas I, Pulli A, Nandy SK, Narayan R (2014) A framework for post-silicon realization of arbitrary instruction extensions on reconfigurable data-paths. J Syst Archit 60(7):592–614

Dongarra J, Gates M, Haidar A, Kurzak J, Luszczek P, Wu P, Yamazaki I, Yarkhan A, Abalenkovs M, Bagherpour N, Hammarling S, Šístek J, Stevens D, Zounon M, Relton SD (2019) Plasma: parallel linear algebra software for multicore using OpenMP. ACM Trans Math Softw 45(2):16:1–16:35

Dongarra JJ, Luszczek P (2011) PLASMA. In: Padua DA (ed) Encyclopedia of parallel computing. Springer, pp 1568–1570

Goetting E, Schultz D, Parlour D, Frake S, Carpenter R, Abellera C, Leone B, Marquez D, Palczewski M, Wolsheimer E, Hart M, Look K, Voogel M, West G, Tong V, Chang A, Chung D, Hsieh W, Farrell L, Carter W (1995) A sea-of-gates FPGA. In: Proceedings ISSCC '95 – international solid-state circuits conference, pp 110–111

Higham NJ (1993) Handbook of writing for the mathematical sciences. SIAM, Philadelphia

Jaiyeoba W, Elyasi N, Choi C, Skadron K (2023) Acts: a near-memory FPGA graph processing framework. In: Proceedings of the 2023 ACM/SIGDA international symposium on field programmable gate arrays, FPGA'23. Association for Computing Machinery, New York, pp 79–89

Kestur S, Davis JD, Williams O (2010) Blas comparison on FPGA, CPU and GPU. In: 2010 IEEE computer society annual symposium on VLSI, pp 288–293

Krüger J, Westermann R (2003) Linear algebra operators for GPU implementation of numerical algorithms. ACM Trans Graph 22(3):908–916

Mahadurkar M, Merchant F, Maity A, Vatwani K, Munje I, Gopalan N, Nandy SK, Narayan R (2014) Co-exploration of NLA kernels and specification of compute elements in distributed memory CGRAs. In: XIVth international conference on embedded computer systems: architectures, modeling, and simulation, SAMOS 2014, Agios Konstantinos, Samos, 14–17 July 2014. IEEE, pp 225–232

De Matteis T, de Fine Licht J, Hoefler T (2020) FBLAS: streaming linear algebra on FPGA. In: SC20: international conference for high performance computing, networking, storage and analysis, pp 1–13

Merchant F, Chattopadhyay A, Garga G, Nandy SK, Narayan R, Gopalan N (2014) Efficient QR decomposition using low complexity column-wise givens rotation (CGR). In: 2014 27th international conference on VLSI design, VLSID 2014, and 2014 13th international conference on embedded systems, Mumbai, 5–9 Jan 2014. IEEE Computer Society, pp 258–263

Merchant F, Maity A, Mahadurkar M, Vatwani K, Munje I, Madhava Krishna C, Sivanandan N, Gopalan N, Raha S, Nandy SK, Narayan R (2015) Micro-architectural enhancements in distributed memory CGRAs for LU and QR factorizations. In: 28th International Conference on VLSI Design, VLSID 2015, Bangalore, 3–7 Jan 2015. IEEE Computer Society, pp 153–158

Merchant F, Vatwani T, Chattopadhyay A, Raha S, Nandy SK, Narayan R (2018) Efficient realization of householder transform through algorithm-architecture co-design for acceleration of QR factorization. IEEE Trans Parallel Distrib Syst 29(8):1707–1720

Nechi A, Groth L, Mulhem S, Merchant F, Buchty R, Berekovic M (2023) FPGA-based deep learning inference accelerators: where are we standing? ACM Trans Reconfigurable Technol Syst 16(4):60:1–60:32

Nimmy J, Ramesh Reddy C, Varadarajan K, Alle M, Fell A, Nandy SK, Narayan R (2008) RECONNECT: a NoC for polymorphic ASICs using a low overhead single cycle router. In: 19th IEEE international conference on application-specific systems, architectures and processors, ASAP 2008, 2–4 July 2008, Leuven. IEEE Computer Society, pp 251–256

Rákossy ZE, Merchant F, Acosta-Aponte A, Nandy SK, Chattopadhyay A (2014a) Efficient and scalable CGRA-based implementation of column-wise givens rotation. In: IEEE 25th international conference on application-specific systems, architectures and processors, ASAP 2014, Zurich, 18–20 June 2014. IEEE Computer Society, pp 188–189

Rákossy ZE, Merchant F, Acosta-Aponte A, Nandy SK, Chattopadhyay A (2014b) Scalable and energy-efficient reconfigurable accelerator for column-wise givens rotation. In: Garcia L (ed) 22nd International conference on very large scale integration, VLSI-SoC, Playa del Carmen, Mexico, 6–8 Oct 2014. IEEE, pp 1–6

Staudigl F, Merchant F, Leupers R (2022) A survey of neuromorphic computing-in-memory: architectures, simulators, and security. IEEE Des Test 39(2):90–99

Tan L, Yan M, Ye X, Fan D (2022) HetGraph: a high performance CPU-CGRA architecture for matrix-based graph analytics. In: Proceedings of the great lakes symposium on VLSI 2022, GLSVLSI '22. Association for Computing Machinery, New York, pp 387–391

Vreca J, Sturm KJX, Gungl E, Merchant F, Bientinesi P, Leupers R, Brezocnik Z (2020) Accelerating deep learning inference in constrained embedded devices using hardware loops and a dot product unit. IEEE Access 8:165913–165926

Part III
Multicore and Reconfigurable Architectures

Field-Programmable Gate Array Architecture

13

Andrew Boutros and Vaughn Betz

Contents

Introduction 418
Methodology and Tools for FPGA Architecture Evaluation 420
Key FPGA Applications 422
Programmable Logic Blocks 423
Programmable Routing 430
Programmable IO 435
Programmable Clock Distribution Networks 437
On-chip Memory 439
DSP Blocks 447
Processor Subsystems 453
System-Level Interconnect: Network-on-Chip 455
Interposers 457
Configuration and Security 459
Conclusion 460
References 460

Abstract

Since their inception more than thirty years ago, field-programmable gate arrays (FPGAs) have grown more complex, more capable, and more diverse in their applications. FPGAs can be reprogrammed at a fundamental level, changing the function and interconnection of millions of elements. By reconfiguring their hardware to match the application, FPGAs often achieve higher energy efficiency, lower latency or faster time-to-market across a very wide range of application domains. A modern FPGA combines many components, from logic

A. Boutros · V. Betz (✉)
Department of Electrical and Computer Engineering (ECE), University of Toronto, Toronto, ON, Canada
e-mail: andrew.boutros@mail.utoronto.ca; vaughn@eecg.utoronto.ca; vaughn@ece.utoronto.ca

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_49

blocks, programmable routing and memory blocks to networks-on-chip and processor subsystems. For best efficiency, each component must be carefully architected to match the needs of a wide range of applications, and to mesh well with the other components. Their design involves many different choices starting from the high-level architectural parameters down to the transistor-level implementation details. This chapter describes the evolution of these FPGA components, their design principles and implementation challenges.

Keywords

FPGA architecture · Reconfigurable computing · Programmable logic · FPGA applications

Introduction

The idea of reconfigurable computing originated in the 1960s with the Fixed Plus Variable Structure Computer (Estrin 1960), which aimed to enhance a conventional computing system with the capability to *temporarily* morph into a more application-specialized architecture. This would be achieved using additional programmable logic and interconnect circuitry to implement operations beyond the capabilities of the fixed datapath processor. A variety of research efforts subsequently investigated ideas for reconfigurable computer architectures that could combine both software-like flexibility and hardware efficiency. However, it was over 20 years later that the first commercially successful reconfigurable computing device, known as a field-programmable gate array (FPGA), was created by Xilinx in 1985.

As illustrated in Fig. 1, FPGAs consist of a two-dimensional array of programmable blocks (logic, IO, and others) that can be flexibly connected using a network of pre-fabricated wires with programmable switches between them. The functionality of all the FPGA blocks and the connectivity of routing switches are controlled using millions of configuration bits, usually stored in static random access memory (SRAM) cells, that can be configured to implement arbitrary digital circuits. The designer describes the desired functionality in a hardware description language (HDL) such as Verilog/VHDL or possibly uses high-level synthesis to translate a higher-level programming language (e.g., C++ or OpenCL) to HDL. The HDL design is then compiled using a complex computer-aided design (CAD) flow into the *bitstream* file, analogous to a software program executable, which is used to program all the FPGA's configuration SRAM cells.

FPGAs combine aspects of general-purpose processors and application-specific integrated circuits (ASICs). Their programmability allows a single FPGA to implement many different applications similar to a software-programmable processor, while the fact that their *hardware* is reconfigurable enables custom systems similar to an ASIC. However, FPGAs have a significantly lower non-recurring engineering cost and shorter time-to-market since they do not require the physical design, layout, fabrication, and verification stages that a custom ASIC would normally go through.

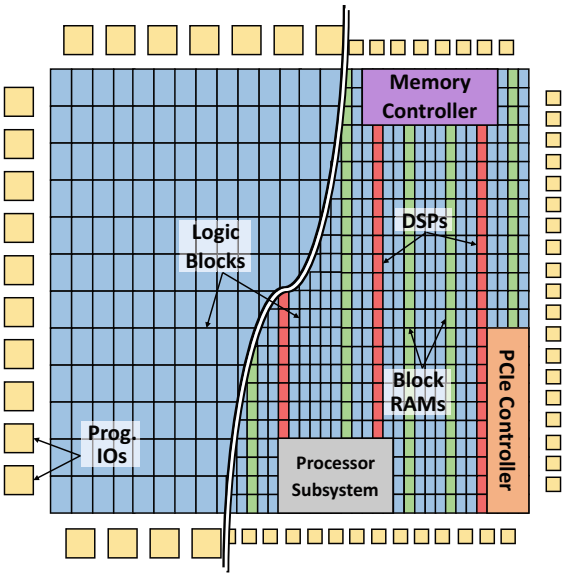


Fig. 1 Early FPGA architecture with programmable logic and IOs vs. modern heterogeneous FPGA architecture with RAMs, DSPs, and other hard blocks. All blocks are interconnected using bit-level programmable routing

A pre-fabricated off-the-shelf FPGA can be used to implement a complete system in a matter of weeks, and also enables continuous hardware upgrades to support new features or fix bugs by simply loading a new bitstream after deployment in-field, thus the name *field-programmable*. This makes FPGAs a compelling solution for medium and small volume designs, especially with the fast-paced product cycles in today's markets. FPGAs can also implement the exact hardware needed for each application (e.g., datapath bitwidth, pipeline stages, number of parallel compute units, memory subsytem, etc.) instead of the fixed one-size-fits-all architecture of general-purpose processors (CPUs) or graphics processing units (GPUs). Consequently, they can achieve higher efficiency than CPUs or GPUs by implementing instruction-free streaming hardware (Hall and Betz 2020) or a processor *overlay* with an application-customized pipeline and instruction set (Boutros et al. 2020).

However, the flexibility of FPGA hardware comes with an efficiency cost compared to ASICs. Kuon and Rose (2007) show that circuits using only the FPGA's programmable logic blocks average 35× larger and 4× slower than corresponding ASIC implementations. A more recent study (Boutros et al. 2018) shows that for full-featured designs which heavily utilize the other FPGA blocks (e.g., RAMs and DSPs), this area gap is reduced to 9×. FPGA architects seek to reduce this efficiency gap as much as possible while maintaining the programmability that makes FPGAs useful across a wide range of applications.

This chapter introduces key principles of FPGA architecture and highlights the evolution of these devices over the past three decades. Figure 1 shows how FPGAs evolved from simple arrays of programmable logic and IO blocks to complex heterogeneous multi-die systems with embedded block RAMs (BRAMs), digital

signal processing (DSP) blocks, processor subsystems, diverse high-performance external interfaces, system-level interconnect, and more. The chapter first gives a brief overview of the CAD flows and methodology used to evaluate new FPGA architecture ideas, as well as key applications for FPGAs. Next, the design principles and challenges for each of the key components of an FPGA architecture are detailed, along with major innovations and future challenges.

Methodology and Tools for FPGA Architecture Evaluation

Figure 2 shows a simplified view of the FPGA architecture evaluation flow. It consists of three main components: a set of benchmark applications, an architecture model, and a CAD system. Unlike an ASIC built for a specific functionality, an FPGA is a general-purpose device that is designed for many use cases, some of which may not even exist when the FPGA is architected. Therefore, a candidate FPGA architecture is evaluated based on its efficiency when used to implement a variety of benchmark designs that are representative of the key FPGA markets and application domains. Typically, each FPGA vendor has a carefully selected set of benchmark designs collected from proprietary system implementations and different customer applications. There are also several open-source benchmark suites such as the classic MCNC20, the VTR (Murray et al. 2020a), and the Titan23 (Murray et al. 2013) suites which are commonly used in academic FPGA architecture and CAD research. While early academic FPGA research used the MCNC suite of designs, these circuits are now too small (thousands of logic primitives) and simple (only IOs and logic) to represent modern FPGA applications. The VTR and particularly the Titan suites are larger and more complex, making them more representative. As FPGA capacity grows and key applications change, new benchmark suites will continue to be needed to drive both FPGA architecture and CAD research.

The second component of the evaluation flow is the FPGA architecture model. The design of an FPGA involves many different decisions from architecture-

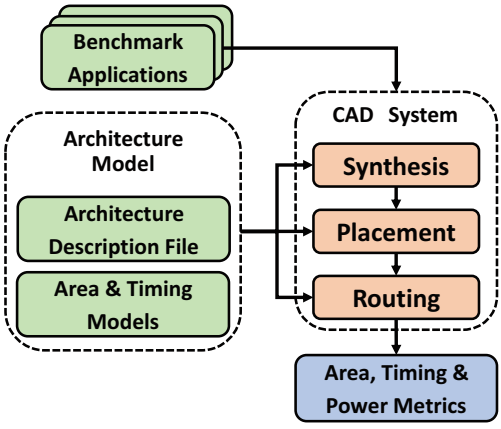


Fig. 2 FPGA architecture evaluation flow

level organization (e.g., number and type of blocks, distribution of wire segment lengths, size of logic clusters and logic elements), to micro-architectural details (e.g., DSP and BRAM modes of operation, hard arithmetic in logic blocks, switch block patterns), and down to transistor-level circuit implementation (e.g., programmable switch type, routing buffer transistor sizing, register implementation). It also involves different implementation styles; the logic blocks and programmable routing are designed and laid out as full-custom circuits, while most hardened blocks (e.g., DSPs) mix standard-cell and full-custom design for the block core and peripherals, respectively. Some blocks, such as BRAMs and high-speed IO, even include significant analog circuitry. All these different components need to be carefully modeled to evaluate the FPGA architecture in its entirety. Tools such as COFFE (Yazdanshenas and Betz 2019) were developed to automate the transistor-level design and modeling of FPGA blocks and programmable routing components, speeding up FPGA architecture investigations. The area, timing, and power models for each of these components are then typically combined in an architecture description file, along with a specification of how these blocks and routing components are organized into the overall architecture.

Finally, a re-targetable CAD system such as the Verilog-to-Routing (VTR) flow (Murray et al. 2020a) is used to map the selected benchmarks to the described FPGA architecture. Such a CAD system consists of a sequence of complex optimization algorithms that synthesizes a benchmark written in an HDL into a circuit netlist, maps it to the different FPGA blocks, places the mapped blocks at specific locations on the FPGA, and routes the connections between them using the specified programmable routing architecture. The implementation produced by the CAD system is then used to evaluate several key metrics. Total area is the sum of the areas of the FPGA blocks used by the application, along with the programmable routing included with them. A timing analyzer finds the critical path(s) through the blocks and routing to determine the maximum frequencies of the application's clocks. Power consumption is estimated based on resources used and signal toggle rates. FPGAs are never designed for only one application, so these metrics are averaged across all the benchmarks. Finally, the overall evaluation blends these average area, delay, and power metrics appropriately depending on the architecture goal (e.g., high performance or low power). Other metrics such as CAD tool runtime and routability of the benchmarks on a candidate architecture are also often considered.

As an example, a key set of questions in FPGA architecture is: What functionality should be *hardened* (i.e., implemented as a new ASIC-style block) in the FPGA architecture? How flexible should this block be? How much of the FPGA die area should be dedicated to it? Ideally, an FPGA architect would like the hardened functionality to be usable by as many applications as possible at the least possible silicon cost. An application that can make use of the hard block will benefit by being smaller, faster and more power-efficient than a *soft* implementation that uses only the programmable logic and routing. This motivates having more programmability in the hard block to capture more use cases; however, higher flexibility generally comes at the cost of larger area and reduced efficiency of the hard block. On the other hand, if a hard block is not usable by an application circuit, its silicon area is

wasted; the FPGA user would rather have more of the usable general-purpose logic blocks in the area of the unused hard block. The impact of this new hard block on the programmable routing must also be considered – does it need more interconnect or lead to slow routing paths to and from the block? To evaluate whether a specific functionality should be hardened or not, both the cost and gain of hardening it have to be quantified empirically using the flow described in this section. FPGA architects may try many ideas before landing on the right combination of design choices that adds just the right amount of programmability to make this new hard block a net win.

In the rest of this chapter, we detail many different components of FPGAs and key architecture questions for each. While we describe the key results without detailing the experimental methodology used to find them, in general they came from a holistic architecture evaluation flow similar to that in Fig. 2.

Key FPGA Applications

In this section, we present some of the key application domains of FPGAs and highlight their advantages for use cases in these areas.

Wireless communications and (e.g., cell phone base stations) is a very large market for FPGAs. The **reconfigurability** of FPGAs allows service providers to implement a range of different standards and upgrade them in-field, while achieving much higher **energy efficiency** compared to instruction-set-based DSP devices. One of the key components in wireless communications is signal processing, such as filtering. The **direct hardware execution** (without an instruction stream) of FPGAs makes them very efficient for repetitive tasks of this nature. Table 1 compares the performance, power and energy efficiency of a Stratix IV FPGA to two Texas Instruments DSP devices (scaled optimistically to the same 40 nm process technology of the FPGA) when implementing simple signal filtering using a 51-tap finite impulse response (FIR) filter. The results show that even a single instance of the FIR filter (consuming only 2% of the FPGA resources) can achieve two orders of magnitude higher performance compared to both DSPs, and 7.7× and 63.2× higher energy efficiency compared to the C5505 and C674x, respectively. The FPGA can achieve another order of magnitude higher performance by instantiating up to 49 instances of the FIR filter working in parallel at the cost of only 9× higher power consumption since the FPGA static power (80% of the FPGA power in Table 1) remains the same regardless of the amount of utilized resources.

Table 1 Performance, power, and energy efficiency comparison between a Stratix IV FPGA and two DSP devices. The results of the DSPs are optimistically scaled to the FPGA's 40 nm process technology

Device	Performance (Samples/s)	Power (mW)	Energy efficiency (Samples/W)
TI C5505	1.77×10^6	28	6.32×10^7
TI C674x	3.21×10^6	416	7.72×10^6
Stratix IV GX230	5.1×10^8	1046	4.88×10^8

Wired communications and networking are also heavy users of FPGAs. The **richness and diversity of FPGA IOs** are important in this use case, as many different and very high-speed IO standards are used in chip-to-chip, server-to-server and city-to-city communications. FPGAs are often used in high-end packet processing and switching systems, which have a high degree of parallelism and a need for high throughput and low latency (Zhao et al. 2020). This is well-suited to an FPGA's **spatial architecture** and the **ability to customize processing pipelines** to the bare minimum required by the target application to reduce latency compared to general-purpose processors with a fixed pipeline and memory hierarchy. The **hardened network transceivers** in modern FPGAs along with the ability to customize the network stack implementation also make FPGAs suitable for ultra-low latency networking interfaces. This can also be useful in other domains, including financial applications such as high-frequency trading (Lockwood et al. 2012) where FPGA reprogrammability allows integration of the rapidly changing trading algorithms on the same chip as the low-latency networking interface.

More recently, FPGAs have also been deployed on a large scale in datacenters where both their computation and networking capabilities are leveraged. The Microsoft Catapult project couples every CPU server with an FPGA that can be used to accelerate search engines, packet processing, encryption and compression (Putnam et al. 2014; Caulfield et al. 2016). This achieved a 95% improvement in the ranking throughput of their search engine infrastructure at the cost of only 10% higher power consumption. The network-connected FPGAs in the Catapult project were also used to implement Brainwave, a datacenter-scale deep learning accelerator for real-time low-latency inference (Fowers et al. 2018).

The **hardware reprogrmmability** of FPGAs has led to their extensive use in ASIC prototyping (Krupnova and Saucier 2000), where either a part or the entirety of an ASIC design is emulated on FPGAs to verify functionality or estimate performance before fabrication. There are a myriad of other application domains for FPGAs including embedded real-time video processing in autonomous vehicles (Rettkowski et al. 2017), genomics (Turakhia et al. 2018), biophotonic simulations (Young-Schultz et al. 2020), accelerated RTL simulation (Karandikar et al. 2018), and many more.

These diverse applications are enabled by the various components of an FPGA architecture working together, and in the following sections we detail the architecture of each of these components.

Programmable Logic Blocks

A fundamental component of an FPGA is the programmable logic block that can implement arbitrary logic functions.

The earliest reconfigurable computing devices were *programmable array logic* (PAL) architectures. As shown in Fig. 3, PALs consisted of an array of `AND` gates feeding another array of `OR` gates which could implement any Boolean logic expression as a two-level sum-of-products function. Programmable switches are

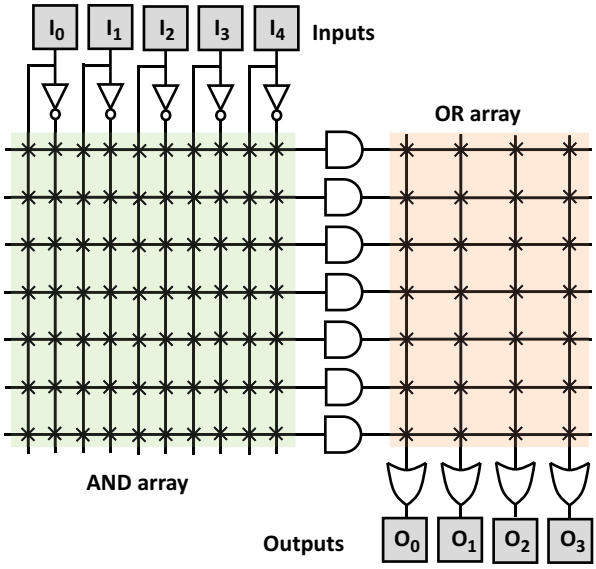


Fig. 3 Programmable array logic (PAL) architecture with an AND array followed by an OR array. The crosses are reconfigurable switches that are programmed to implement any Boolean expression as a two-level sum-of-products function

flexibly configured to select the inputs to each of the AND/OR gates to implement different Boolean expressions. The design tools for PALs were very simple since the delay through the device is constant regardless of the logic function implemented. However, PALs did not scale well; as device logic capacity increased, the wires connecting the AND/OR grid became increasingly longer and slower and the number of required programmable switches grew quadratically.

Subsequently, *complex programmable logic devices* (CPLDs) kept the AND/OR arrays as the basic logic elements, but attempted to solve the scalability challenge by integrating multiple PALs on the same die with a crossbar interconnect between them at the cost of more complicated design tools. Shortly after, Xilinx pioneered the first FPGA in 1985, which consisted of an array of SRAM-based lookup tables (LUTs) with programmable interconnect between them. This style of reconfigurable devices was shown to scale very well, with LUTs achieving much higher area efficiency compared to the AND/OR logic in PALs and CPLDs. Consequently, LUT-based architectures became increasingly dominant and today LUTs form the fundamental logic element in all commercial FPGAs. Some research attempts investigated replacing LUTs with a different form of configurable AND gates: a full binary tree of AND gates with programmable output/input inversion known as an *AND-inverter cone* (AIC) (Parandeh-Afshar et al. 2012). However, when thoroughly evaluated in Zgheib et al. (2014), AIC-based FPGA architectures had significantly larger area than LUT-based ones, with delay gains only on small benchmarks that have relatively short and localized critical paths.

A K-LUT can implement any K-input Boolean function by storing its truth table in 2^K configuration SRAM cells. Figure 4a shows the transistor-level circuit implementation of a 4-LUT using pass-transistor logic. The four inputs (A, B, C, and D) are used as multiplexer select lines to choose an output from the 16 values

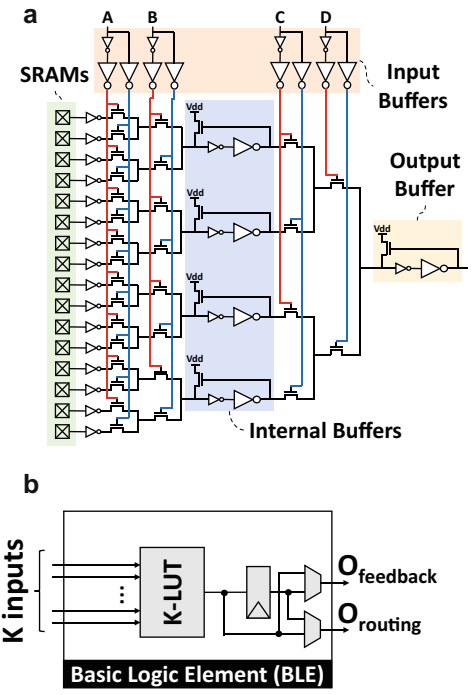


Fig. 4 (**a**) Transistor-level implementation of a 4-LUT with internal buffers between the second and third LUT stages, and (**b**) Basic logic element (BLE)

of the truth table in the SRAMs. In addition to the output buffer, an internal buffering stage (shown between the second and third stages of the LUT in Fig. 4a) is typically implemented to mitigate the quadratic increase in delay when passing through a chain of pass-transistors. The sizing of the LUT's pass-transistors and the internal/output buffers is carefully tuned to achieve the best area-delay product. Classic FPGA literature (Betz et al. 1999) defines the *basic logic element* (BLE) as a K-LUT coupled with an output register and 2:1 bypassing multiplexers as shown in Fig. 4b. Thus, a BLE can be used to implement just a flip-flop (FF) with the LUT configured as an identity function, or any Boolean expression with up to K inputs and optionally-registered output. As illustrated in Fig. 5a, BLEs are typically clustered in *logic blocks* (LBs), such that an LB contains N BLEs along with local interconnect. The local interconnect in the logic block consists of multiplexers between signal sources (BLE outputs and logic block inputs) and destinations (BLE inputs). These multiplexers are often arranged to form a local full (Betz and Rose 1998) or partial (Lemieux et al. 2000) crossbar. At the circuit level, these multiplexers are usually built as two levels of pass transistors, followed by a two-stage buffer as shown in Fig. 5b; this is the most efficient circuit design for FPGA multiplexers in most cases (Chiasson and Betz 2013a). Figure 5a also shows the switch and connection block multiplexers forming the programmable routing used for inter-LB communication; this routing is discussed in detail in the "Programmable Routing" section.

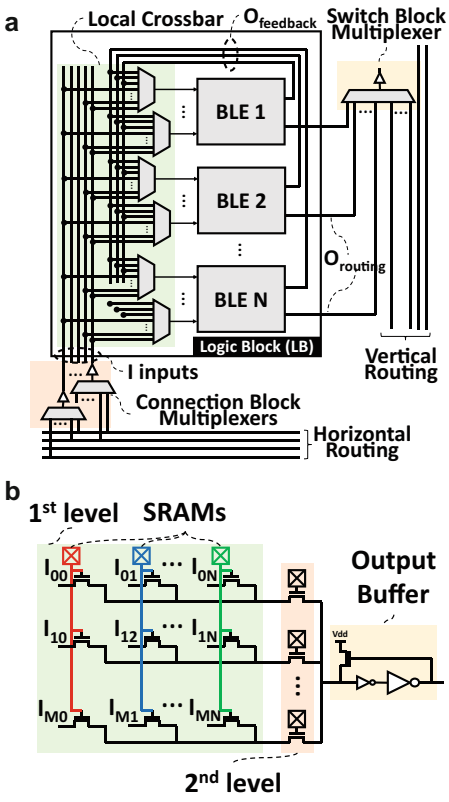


Fig. 5 (**a**) Logic block (LB) internal architecture, and (**b**) Two-level multiplexer circuitry

Over the years, the size of LUTs (K) and LBs (N) have gradually increased with growing device logic capacity. As K increases, more functionality can be captured into a single LUT. Therefore, the same circuit can be implemented using fewer LUTs with a smaller number of logic levels on the critical path, which increases performance. In addition, the demand for inter-LB routing decreases as more connections are captured into the fast local interconnect by increasing N. On the other hand, the area of the LUT increases exponentially with K (due to the 2^K SRAM cells) and its speed degrades linearly (due to propagation through a chain of K pass transistors with periodic buffering). The size of the local crossbar also increases quadratically and its speed degrades linearly with increasing N. Ahmed and Rose (2004) empirically evaluated these trade-offs and found that LUTs of size 4–6 and LBs of size 3–10 BLEs offer the best area-delay product for an FPGA architecture, with 4-LUTs leading to a better area but 6-LUTs yielding a higher speed. Historically, the first Xilinx FPGAs had an LB with only two 3-LUTs (i.e., $N = 2$, $K = 3$). LB size gradually increased over time and by 1999, Xilinx's Virtex family had four 4-LUTs and Altera's Apex 20K family had ten 4-LUTs in each LB.

The next major logic feature was the *fracturable* LUTs introduced in 2003 by Altera in their Stratix II architecture. Ahmed and Rose in (2004) showed that

an LB with ten 6-LUTs achieved 14% better performance but increased area by 17% compared to an LB with ten 4-LUTs. In addition, an architecture with only 6-LUTs can suffer from significant under-utilization. Lewis et al. found that 64% of the LUTs implemented for a commercial benchmark suite used fewer than 6 inputs, wasting some of the 6-LUT functionality (Lewis et al. 2005). Based on these observations, fracturable LUTs were introduced to combine the best of both worlds: the higher performance of larger LUTs and the superior area-efficiency of smaller ones. A fracturable $\{K, M\}$-LUT can be configured as a single LUT of size K or can be fractured into two LUTs of size up to $K-1$ that collectively use no more than $K + M$ distinct inputs. Figure 6a shows that a 6-LUT is internally composed of two 5-LUTs plus a 2:1 multiplexer. Consequently, almost no circuitry (only the red added output) is necessary to allow a 6-LUT to instead operate as two 5-LUTs that share the same inputs. However, this requires the two 5-LUTs to share all their inputs which limits how often both LUTs can be simultaneously used. Adding extra routing ports as shown in Fig. 6b relaxes this constraint and makes it easier to find two logic functions that can be packed together into a fracturable 6-LUT at the cost of slightly increasing its area. The *adaptive logic module* (ALM) in the Stratix II architecture implemented a {6, 2}-LUT that had 8 input and 2 output ports. Thus, an ALM can implement a 6-LUT or two 5-LUTs sharing 2 inputs (and therefore a total of 8 distinct inputs). Pairs of smaller LUTs could also be implemented without any shared inputs, such as two 4-LUTs or one 5-LUT and one 3-LUT. With a fracturable 6-LUT, larger logic functions are implemented in 6-LUTs reducing the logic levels on the critical path and achieving better performance. On the other hand, pairs of smaller logic functions can be packed together (each using only half an ALM), improving area-efficiency. The LB in Stratix II not only increased the performance by 15%, but also reduced the logic and routing area by 2.6% compared to a baseline 4-LUT-based LB (Lewis et al. 2005).

Xilinx later adopted a similar approach in their Virtex-5 architecture in which the 6-LUTs can also be decomposed into two 5-LUTs. However, they adopted a LUT architecture similar to that shown in Fig. 6a with minimal changes compared to the traditional 6-LUT (i.e., no extra input routing ports or steering multiplexers).

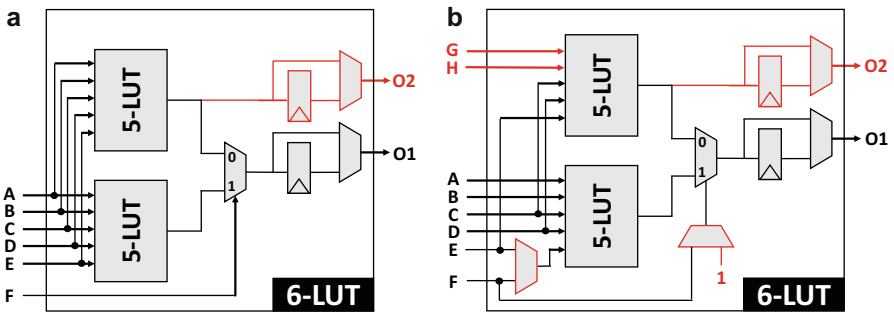


Fig. 6 6-LUT fracturable into two 5-LUTs with (**a**) no additional input ports, leading to 5 shared inputs or (**b**) two additional input ports and steering multiplexers, leading to only 2 shared inputs

This results in a lower area per fracturable LUT, but makes it more difficult to pack two smaller LUTs together as they must use no more than 5 distinct inputs. While subsequent architectures from both Altera/Intel and Xilinx have also been based on fracturable 6-LUTs, recent work from Microsemi (Feng et al. 2018) revisited the 4-LUT vs. 6-LUT efficiency trade-off for newer process technologies, CAD tools and designs than those used in Ahmed and Rose (2004). It shows that a LUT structure with two tightly coupled 4-LUTs, one feeding the other, can achieve performance close to conventional 6-LUTs while maintaining the high utilization and area efficiency of 4-LUTs. In terms of LB size, FPGA architectures from Altera/Intel and Xilinx converged on the use of relatively large LBs with ten and eight BLEs respectively, for several generations. However, the Versal architecture from Xilinx further increases the number of BLEs per LB to thirty two (Gaide et al. 2019). This significant increase in LB size is motivated by two main factors. First, inter-LB wire delay is scaling poorly with process shrinks, so capturing more connections within an LB's local routing is increasingly beneficial. Second, ever-larger FPGA designs tend to increase CAD tool runtime, but larger LBs can mitigate this trend by simplifying placement and inter-LB routing.

The number of FFs per BLE and the circuit-level FF implementation are other important architecture choices. Early FPGAs with non-fracturable LUTs had a single FF to optionally register the LUT output as shown in Fig. 4b. When they moved to fracturable LUTs, both Altera/Intel and Xilinx architectures added a second FF to each BLE so that both outputs of the fractured LUT could be registered, as shown in Fig. 6a and b. In the Stratix V architecture, the number of FFs was doubled (i.e., four FFs per BLE) to accommodate the increasing demand for FFs as designs became more deeply pipelined to achieve higher performance (Lewis et al. 2013). Low-cost multiplexing circuitry allows sharing the existing inputs between the LUTs and FFs to avoid adding more costly routing ports. Stratix V also implements FFs as pulse latches instead of edge-triggered FFs. As shown in Fig. 7b, this removes one of the two latches that would be present in a master-slave FF (Fig. 7a), reducing the register delay and area. A pulse latch acts as a cheaper FF with worse hold time as it latches the data input during a very short pulse instead of a clock edge as in conventional FFs. However, it would be area-inefficient to build a pulse generator for each latch. Instead, this cost is amortized

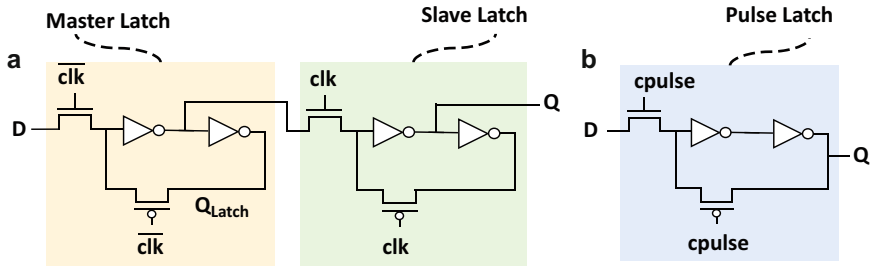


Fig. 7 Circuitry for (**a**) Master-slave positive-edge-triggered FF, and (**b**) Pulse latch

by implementing only two configurable pulse generators per LB; each of the 40 pulse latches in an LB selects which generator provides its pulse input. The FPGA CAD tools can also program the pulse width in these generators, allowing a limited amount of time borrowing between source and destination registers. Soon after, the Xilinx Ultrascale+ architecture also adopted the use of pulse latches as its FFs due to their area and speed benefits (Ganusov and Devlin 2016).

Murray et al. found that 22% of logic elements in the Titan suite of benchmarks implemented addition or subtraction (Murray et al. 2020b). When implemented with LUTs, each bit of arithmetic in a ripple carry adder requires two LUTs, one for generating the sum and another for the carry. This is inefficient as it results in high logic utilization and a slow critical path due to having many cascaded LUTs in series for computing the carries in multi-bit additions. Consequently, all modern FPGA architectures include hardened arithmetic circuitry in their LBs. There are many variants, but all have several common points. First, to avoid adding expensive routing ports, the arithmetic circuits re-use the LUT routing ports or are fed by the LUT outputs. Second, the carry bits are propagated on a special, dedicated interconnect with little or no programmability so that the crucial carry path is fast. The lowest cost arithmetic circuitry hardens ripple carry structures and achieves a large speed gain over LUTs (3.4× for a 32-bit adder in Murray et al. 2020b). Hardening more sophisticated structures like carry skip adders further improves speed (an additional 20% speed-up at 32 bits in Yazdanshenas and Betz 2019). The latest Versal architecture from Xilinx (Gaide et al. 2019) hardens the carry logic for 8-bit carry look-ahead adders (i.e., the addition can only start on every eighth BLE), while the sum, propagate and generate logic is all implemented in the fracturable 6-LUTs feeding the carry logic as shown in Fig. 8a. This organization allows implementing 1-bit of arithmetic per logic element. On the other hand, the latest Intel Agilex architecture can implement two bits of arithmetic per logic

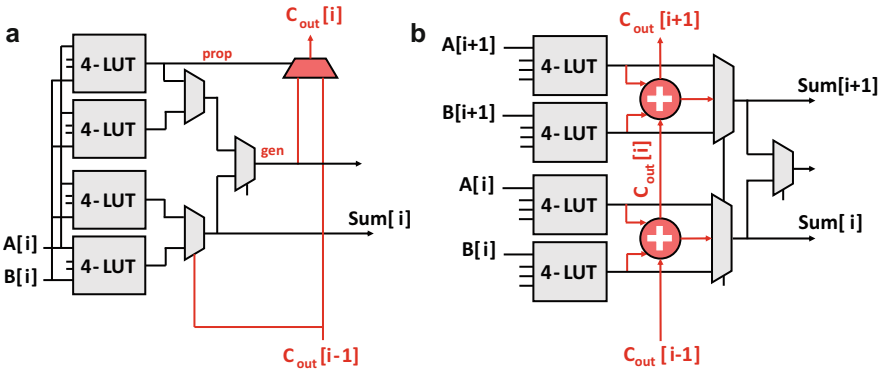


Fig. 8 Overview of the hard arithmetic circuitry (in red) in the logic elements of (**a**) Xilinx and (**b**) Altera/Intel FPGAs. $A[i]$ and $B[i]$ are the ith bits of the two addition operands A and B. The Xilinx logic elements compute carry propagate (`prop`) and generate (`gen`) in the LUTs, while the Altera/Intel ones use LUTs to pass inputs to the hard adders. Unlabeled inputs are unused when implementing adders

element, with a dedicated interconnect for the carry as shown in Fig. 8b. It achieves that by hardening 2-bit carry-skip adders that are fed by the four 4-LUTs contained within a fracturable 6-LUT (Chromczak et al. 2020). The study by Murray et al. (2020b) shows that the combination of fracturable LUTs and 2 bits of arithmetic (similar to that adopted in Altera/Intel FPGAs) is particularly efficient compared to architectures with non-fracturable LUTs or 1 bit of arithmetic per logic element. It also concludes that having dedicated arithmetic circuits (i.e., hardening adders and carry chains) inside the FPGA logic elements increases average performance by 75% and 15% for arithmetic microbenchmarks and general benchmark circuits, respectively.

Recently, deep learning (DL) has become a key workload in many end-user applications, with its core operation being multiply-accumulate (MAC). Generally, MACs can be implemented in DSP blocks as will be described in the "DSP Blocks" section; however, low-precision MACs with 8-bit or narrower operands (which are becoming increasingly popular in DL workloads) can also be implemented efficiently in the programmable logic (Caulfield et al. 2016). In this case, LUTs implement `AND` gates to generate partial products followed by an adder tree to reduce the partial products and perform the accumulation. Consequently, multiple recent studies (Rasoulinezhad et al. 2020; Eldafrawy et al. 2020) have investigated increasing the density of hardened adders in the FPGA's logic fabric to enhance its performance when implementing arithmetic-heavy applications such as DL. The work in Eldafrawy et al. (2020) proposed multiple different logic block architectures that incorporate 4 bits of arithmetic per logic element arranged in one or two carry chains with different configurations, instead of just 2 bits of arithmetic in an Intel Stratix-like ALM. These proposals do not require increasing the number of the (relatively expensive) routing ports in the logic clusters when implementing multiplications due to the high degree of input sharing in a multiplier array (i.e., for an N-bit multiplier, only $2N$ inputs are needed to generate N^2 partial products). The most promising of these proposals increases the density of MAC operations by $1.7\times$ while simultaneously improving their speed. It also reduces the required logic and routing area by 8% for general benchmarks, highlighting that more arithmetic density is beneficial for applications beyond DL.

Programmable Routing

Programmable routing commonly accounts for 50% or more of the fabric area and the critical path delay of applications (Chiasson and Betz 2013b), so its efficiency is crucial. Programmable routing is composed of pre-fabricated wire segments connected via programmable switches. By programming an appropriate sequence of switches to be on, a connection can be formed between any two function blocks. There are two main classes of FPGA routing architecture. *Hierarchical* FPGAs are inspired by the fact that designs are inherently hierarchical; higher-level modules instantiate lower-level modules and connect signals between them, with communication being more frequent between modules that are near each other in the design hierarchy. As shown in Fig. 9, hierarchical FPGAs can realize these

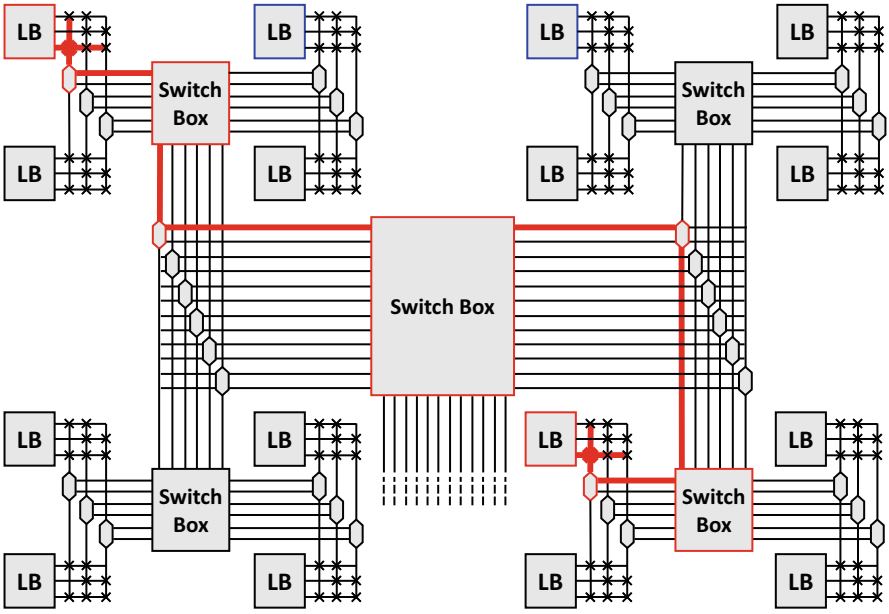


Fig. 9 Hierarchical routing architecture. A distant connection (highlighted in red) traverses through different levels of the hierarchy. Some blocks in physical proximity (highlighted in blue) still require several wires and switches to connect

connections with short wires that connect small regions of the chip. To communicate to more distant regions, a connection (highlighted in red) passes through multiple wires and switches as it traverses different levels of the interconnect hierarchy. This style of architecture was popular in many earlier FPGAs, such as Altera's Flex and Apex families, but it leads to very long wires at the upper levels of the interconnect hierarchy which became problematic as process scaling made such wires increasingly resistive. A strictly hierarchical routing architecture also results in some blocks that are physically close together (e.g., the blue blocks in Fig. 9) which still require several wires and switches to connect. Consequently, this routing architecture is primarily used today for smaller FPGAs, such as the FlexLogix FPGA IP cores that can be embedded in larger SoC designs.

The other type of FPGA interconnect is *island-style*, as depicted in Fig. 10. This architecture was pioneered by Xilinx and is inspired by the fact that a regular two-dimensional layout of horizontal and vertical directed wire segments can be efficiently laid out. As shown in Fig. 10, island-style routing includes three components: routing wire segments, connection blocks (multiplexers) that connect function block inputs to the routing wires, and switch blocks (programmable switches) that connect routing wires together to realize longer routes. The placement engine in FPGA CAD tools chooses which function block implements each element of a design in order to minimize the required wiring. Consequently, most connections between function blocks span a small distance and can be implemented with a few routing wires as illustrated by the red connection in Fig. 10.

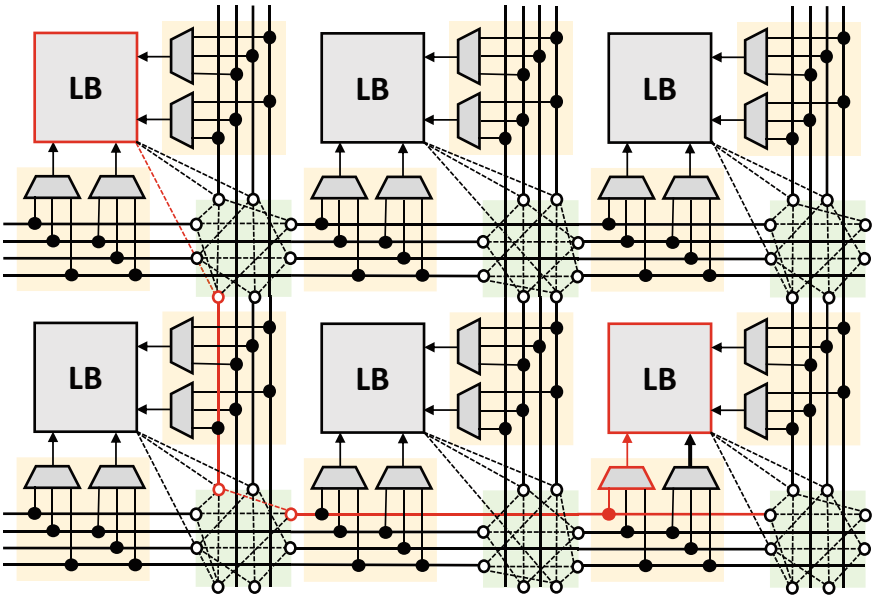


Fig. 10 Island-style routing architecture. Thick solid lines are routing wires while dashed lines are programmable switches. Connection and switch blocks are shaded in yellow and green, respectively

Creating a good routing architecture involves managing many complex trade-offs. It should contain enough programmable switching and wire segments that the vast majority of circuits can be implemented; however, too many wires and switches waste area and complicate the routing CAD problem. A routing architecture should also match the needs of applications. Ideally, short connections will be made with short wires to minimize capacitance and layout area, while long connections can use longer wiring segments to avoid the extra delay of passing through many routing switches. FPGA routing architecture design is also challenging as it involves many different and interacting choices. These choices include: how many routing wires each logic block input or output can connect to (F_c), how many other routing wires each wire can connect to (F_s), the lengths of the routing wire segments, the routing switch pattern, the electrical design of the wires and switches themselves, and the number of routing wires per channel (Betz et al. 1999). In Fig. 10 for example, $F_c = 3$, $F_s = 3$, the channel width is 4 wires, and some routing wires are of length 1, while others are of length 2. Fully evaluating these trade-offs and selecting the values for these architecture parameters for target applications and at a specific process node requires experimentation using a full CAD flow as previously discussed in the "Methodology and Tools for FPGA Architecture Evaluation" section.

Early island-style architectures incorporated only short wires that traversed a single logic block between programmable switches. Later research showed that this

resulted in more programmable switches than necessary, and that making all wiring segments span four logic blocks before terminating reduced application delay by 40% and routing area by 25% (Betz and Rose 1999). Modern architectures include multiple lengths of wiring segments to better match the needs of short and long connections, but the most plentiful wire segments remain of moderate length, with four logic blocks being a popular choice. Longer distance connections can achieve lower delay using longer wire segments, but in recent process nodes wires that span many (e.g., 16) logic blocks must use wide and thick metal traces on upper metal layers to achieve acceptable resistance (Petelin and Betz 2016). The amount of such long-distance wiring one can include in a metal stack is limited. To best leverage such scarce wiring, Intel's Stratix FPGAs allow long wire segments to be connected only to short wire segments, rather than function block inputs or outputs (Lewis et al. 2003). This creates a form of routing hierarchy within an island-style FPGA, where short connections use only the shorter wires, but longer connections pass through short wires to reach the long wire network. Another area where hierarchical FPGA concepts are used within island-style FPGAs is within the logic blocks. As illustrated in Fig. 5a, most logic blocks now group multiple BLEs together with local routing. This means that each logic block is a small cluster in a hierarchical FPGA; island-style routing interconnects the resulting thousands of logic clusters.

There has been a great deal of research into the optimal amount of switching, and how to best arrange the switches. While there are many detailed choices, a few principles have emerged. The first is that the connectivity between function block pins and wires (F_c) can be relatively low: typically only 10% or less of the wires that pass by a pin will have switches to connect to it. Similarly, the number of other wires that a routing wire can connect to at its end (F_s) can also be low, but it should be at least 3 so that a signal can turn left, right, or go straight at a wire endpoint. The local routing in a logic cluster (described in the "Programmable Logic Blocks" section) allows some block inputs and some block outputs to be swapped during routing (i.e., general programmable routing can deliver a signal to one of several input pins, which can then be routed to the right LUT input using the local crossbar). By leveraging this extra degree of flexibility and considering all the options presented by the multi-stage programmable routing network, the routing CAD tool can achieve high completion rates even with low F_c and F_s values. Switch patterns that give more options to the routing CAD tool also help routability; for example, the Wilton switch pattern ensures that following a different sequence of channels lets the router reach different wire segments near a destination block (Tang et al. 2019). Some recent architectures have also created L-shaped routing segments (formed by shorting a horizontal and vertical metal segment together) that allow connections between diagonally nearby blocks with fewer routing switches (Sivaswamy et al. 2005; Petersen et al. 2021).

There are also multiple options for the electrical design of programmable switches, as shown in Fig. 11. Early FPGAs used pass gate transistors controlled by SRAM cells to connect wires. While this is the smallest switch possible, the delay of routing wires connected in series by pass transistors grows quadratically,

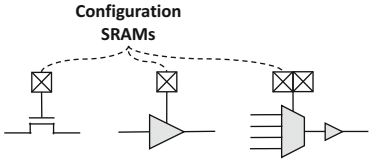


Fig. 11 Different implementations for SRAM-controlled programmable switches using pass transistors (left), tri-state buffers (middle), or buffered multiplexers (right)

making them very slow for large FPGAs. Adding some tri-state buffer switches costs area, but improves speed (Betz and Rose 1999). Most recent FPGAs primarily use a multiplexer built out of pass gates followed by a buffer that cannot be tri-stated, as shown in detail in Fig. 5b. The pass transistors in this *direct drive* switch can be small as they are lightly loaded, while the buffer can be larger to drive the significant capacitance of a routing wire segment. Such direct drive switches create a major constraint on the switch pattern: a wire can only be driven at one point, so only function block outputs and routing wires near that point can feed its routing multiplexer inputs and hence be possible signal sources. Despite this constraint, both academic and industrial work has concluded that direct drive switches improve both area and speed due to their superior electrical characteristics (Lewis et al. 2003; Lemieux et al. 2004). The exception is expensive or rare wires such as long wires implemented on wide metal traces on upper metal layers or the interposer-crossing wires discussed later in the "Interposers" section. These wires often have multiple tri-state buffers that can drive them, as the cost of these larger programmable switches is merited to allow more flexible usage of these expensive wires.

A major challenge for FPGA routing is that the delay of long wires is not improving with process scaling, which means that the delay to cross the chip is stagnating or increasing even as clock frequencies rise. This has led FPGA application developers to increase the amount of pipelining in their designs, thereby allowing multiple clock cycles for long routes. To make this strategy more effective, some FPGA manufacturers have integrated registers within the routing network itself. Intel's Stratix 10 device allows each routing driver (i.e., multiplexer followed by a buffer) to be configured as a pulse latch as shown in Fig. 7b, thereby acting as a register with low delay but relatively poor hold time. This allows deep pipelining of interconnect without using expensive logic resources, at the cost of a modest area and delay increase to the routing driver (Lewis et al. 2016). However, their poor hold time means using pulse latches in immediately consecutive Stratix 10 routing switches would lead to hold time violations, so not all of these interconnect registers can be simultaneously used. Therefore, Intel refined this approach in their Agilex devices by integrating actual registers (with better hold time) on only one-third of the interconnect drivers to mitigate the area cost (Chromczak et al. 2020). Rather than integrating registers throughout the interconnect, Xilinx's Versal devices instead add bypassable registers only on the inputs to function blocks. Unlike Intel's interconnect registers, these input registers are full-featured, with clock enable and clear signals (Gaide et al. 2019).

Since neighboring LB are likely to implement related logic, FPGA architectures also include dedicated interconnects to implement high-speed connections between adjacent LBs. Such connections are realized by allowing the outputs of an LB to drive the local input crossbar of its immediate neighbors without using the general programmable routing. The FPGA CAD tools can then decide how to place the implemented circuit such that critical inter-LB connections can benefit from these dedicated interconnects. To further increase the efficiency of the routing architecture, some recent studies build on this idea by analyzing a variety of benchmark circuits to extract the most commonly used routing patterns and implement them as dedicated routing structures (Nikolić et al. 2020). This results in a modest 3% improvement in the average critical path delay of the studied benchmarks, but these gains can be potentially improved through CAD enhancements to better exploit the dedicated routing.

Programmable IO

One of the unique properties of FPGAs is their programmable IO structures that allow them to communicate with a wide variety of other devices, making them the communications hub of many systems. For a single set of physical IOs to *programmably* support many different IO interfaces and standards, it requires adaptation to different voltage levels, electrical characteristics, timing specifications, and command protocols. Both the value and the challenge of programmable IO are highlighted by the large area devoted to IOs on FPGAs. For example, Altera Stratix II (90 nm) devices devote 20% (largest device) to 48% (smallest device) of their die area to IO-related structures and support 28 different IO standards.

As Fig. 12 shows, FPGAs address the challenges of programmable IO design using a combination of approaches (Tyhach et al. 2004; Qian et al. 2018). First,

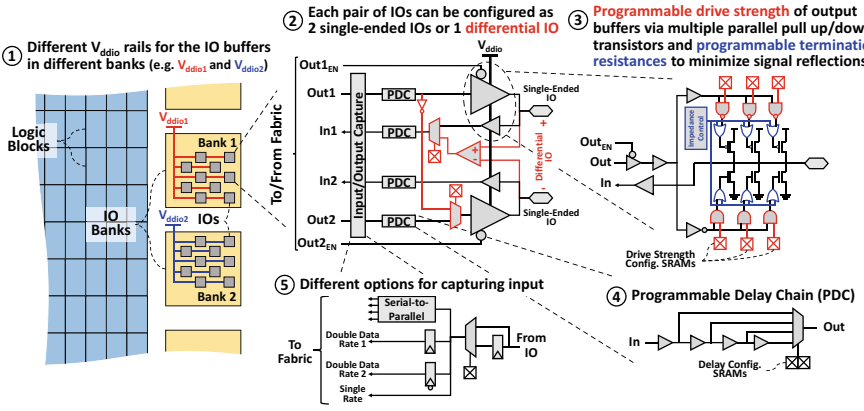


Fig. 12 Overview of the different techniques for implementing programmable IOs in FPGAs

FPGAs use IO buffers that can operate across a range of voltages. As shown in ①, these IOs are grouped into banks (commonly on the order of 50 IOs per bank), where each bank has a separate V_{ddio} rail for the IO buffers. This allows different banks to operate at different voltage levels. For example, IOs in one bank could be operating at 1.8 V while those in a different bank operate at 1.2 V. Second, each IO can be used separately for single-ended standards, or pairs of IOs can be programmed to implement the positive and negative lines for differential IO standards as in ②. Third, IO buffers are implemented with multiple parallel pull-up and pull-down transistors so that their drive strengths can be programmably adjusted by enabling or disabling different numbers of pull-up/pull-down pairs. This is illustrated in part ③ of Fig. 12. By programming some pull-up or pull-down transistors to be enabled even when no output is being driven, FPGA IOs can minimize signal reflections by implementing different on-chip termination resistances. Programmable delay chains, shown in ④, provide a fourth level of configurability, allowing fine delay adjustments of signal timing to and from the IO buffer.

In addition to electrical and timing programmability, FPGA IO blocks contain additional hardened digital circuitry to simplify capturing and transferring IO data to the fabric. Generally, some or all of this hardened circuitry can be bypassed by SRAM-controlled muxes, allowing FPGA users to choose which hardened functions are desirable for a given design and IO protocol. Part ⑤ of Fig. 12 shows a number of common digital logic options on the IO input path: a capture register, double-to-single data rate conversion registers (used with DDR memories), and serial-to-parallel converters to allow transfers to the programmable fabric operating at a lower frequency. Most FPGAs now also contain bypassable blocks that connect to a group of IOs and implement higher-level protocols like DDR memory controllers. Together these approaches allow the general-purpose FPGA IOs to service many different protocols, at speeds up to 3.2 Gb/s.

The highest speed IOs implement serial protocols, such as PCIe and Ethernet, that embed the clock in data transitions and can run at 28 Gb/s or more. To achieve these speeds, FPGAs include a separate group of differential-only IOs that can only be used as serial transceivers and have less voltage and electrical programmability (Upadhyaya et al. 2016). Just as for the general-purpose IOs, these serial IOs have a sequence of high-speed hardened circuits between them and the fabric, some of which can be optionally bypassed to allow end-users to customize the exact interface protocol.

Overall, FPGA IO design is very challenging, due to the dual (and competing) demands to make the IO not only very fast but also programmable. In addition, the rest of the FPGA fabric should also be designed appropriately to keep up with the IO bandwidth; distributing the very high data bandwidths from IO interfaces requires wide *soft* buses to be configured using the programmable routing and logic. This creates additional challenges that will be discussed later in the "System-Level Interconnect: Network-on-Chip" section.

Programmable Clock Distribution Networks

Since FPGA applications are often communicating with many different devices at different speeds, they commonly include many different clock domains. Most of these clocks are generated on-chip by programmable phase-locked loops (PLLs), delay-locked loops (DLLs) and clock data recovery (CDR) circuits. Distributing that many high-speed clocks to all the FFs on the chip using the general programmable routing (discussed in the "Programmable Routing" section) would be extremely challenging for several reasons:

1. Both the programmable routing architecture and the routing CAD algorithms for general signals focus on optimizing delay and wire usage. However, routing clock signals has a different objective: minimizing the clock skew (i.e., balancing the delay) between different endpoints. While specialized low-skew routing CAD algorithms have been devised, they still struggle to create balanced trees in a general programmable interconnect that is not optimized for this case. The difficulty increases for major system clocks, which can have fanouts of hundreds of thousands of registers.
2. The programmable routing wires are optimized for density and speed rather than minimal process variation, and this increases the uncertainty of clocks routed on them, which in turn degrades timing. Another source of increased uncertainty is the capacitive crosstalk between the densely spaced routing wires. A signal (routed on an adjacent wire) toggling at the same time as the clock edge will add significant clock jitter, degrading both setup and hold timing.
3. The very high toggle rate of clocks makes adding extra capacitance to their routing highly undesirable, as it will have a significant power impact. The inefficiency of the general routing wires in creating balanced trees due to both extra switches and suboptimal switch patterns for this case will lead to higher clock capacitance and power consumption.

As a result, FPGAs typically have dedicated interconnect networks for clock distribution, which still have to be flexible enough since the clock domain of each register can vary from one design to another.

Clock networks use routing wires and switch topologies that allow the construction of low-skew networks like *H-trees*. As shown in Fig. 13, these trees have a fractal pattern in the shape of the letter *H* with the signal source at the center and equal delays to reach the four endpoints. These distribution trees minimize clock uncertainty by using wider metal wires with bigger buffers between tree hierarchy levels to minimize process variation and shielded trees to reduce crosstalk-induced jitter. However, an FPGA design can have dozens of clocks, with many of them spanning sub-regions of the chip near the programmable IOs (e.g., a single DDR3 interface typically uses 5–7 different clocks) (Hutton et al. 2005). Pre-fabricating dozens of H-trees that span the entire chip would be one possible clocking architecture, but it would be very expensive, as the lowest level of each of

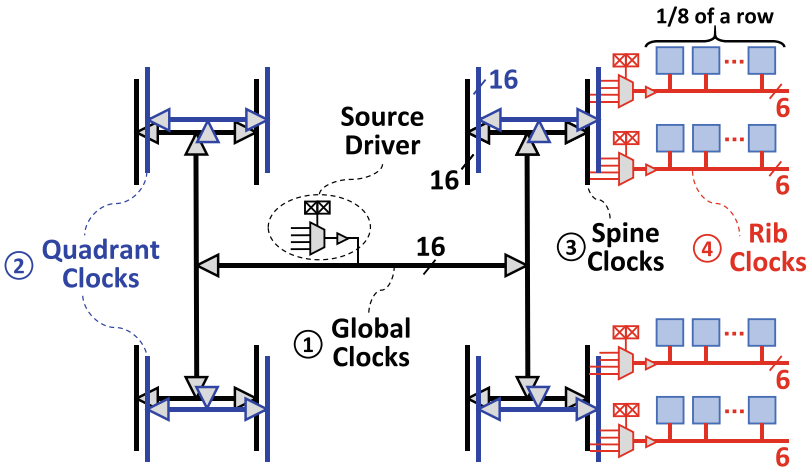


Fig. 13 An example programmable clock distribution network similar to that of Stratix V FPGAs. It has 16 chip-wide global H-trees (black), 16 smaller H-trees per quadrant (blue), and spine-and-ribs leaf distribution (red)

these H-trees would add approximately one (wide and shielded) wire to each routing channel.

Consequently, several techniques are commonly used to implement cheaper clock distribution networks. Since not all clocks are needed everywhere on the chip, some global (chip-wide) H-trees for major clock domains are built along with some smaller ones that cover only portions (e.g., quadrants) of the chip as marked by ① and ② in Fig. 13, respectively. For example, fabricating 16 global and 16 quadrant H-trees enables the use of up to 80 different clocks (16 clocks on the global networks + 4 × 16 clocks on the quadrant clocks) at a cost equivalent to that of only 32 global H-trees. Additional wire savings are achieved by implementing the leaf wiring in a *spine-and-ribs* style as indicated by ③ and ④ in Fig. 13 instead of continuing the H-tree fractal pattern down to individual blocks. The last wire level in an H-tree is called a *spine clock* and it drives several *rib clocks* that each span a fraction of an FPGA row. The clock skew is tolerable as long as the spine and rib wires are kept reasonably short. To further reduce the cost of the leaf wires (ribs) of the clock network, programmable multiplexers are added to select only a portion of the possible spine clock sources to be routed to the rib clocks that functional blocks can access. In Fig. 13 for example, 32 clock trees are multiplexed down to 6 *rib clocks*, reducing the expensive wiring at the leaves of the clock networks by 81%. This multiplexing leads to a constraint: all the function blocks spanned by a rib clock (1/8 of a row in many Altera/Intel FPGAs) must together use no more than 6 distinct clocks. This constraint is enforced automatically by the placement CAD tool during optimization.

The most recent FPGAs have made clocking networks more flexible. In the Intel Stratix 10 architecture, the FPGA chip is divided into clock sectors where

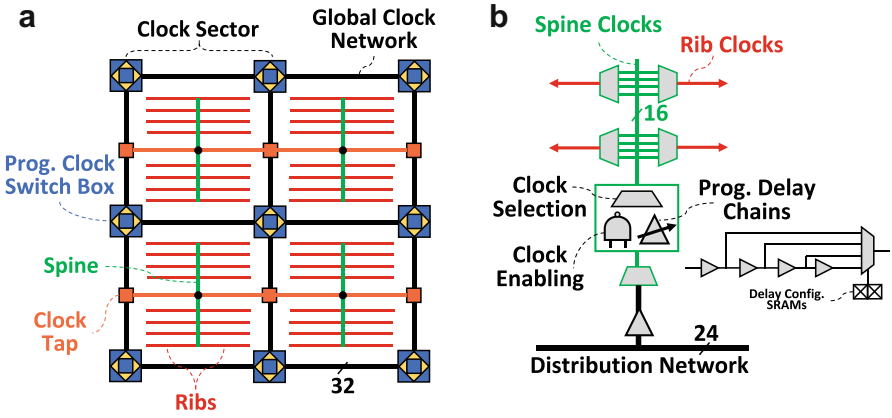


Fig. 14 (**a**) Routable clock networks in Intel Stratix 10 and (**b**) Spine clock control in Xilinx Ultrascale+

traditional spine-and-ribs clock distribution is used within each sector, as shown in Fig. 14a. At the chip level, 32 more flexible distribution networks are implemented with programmable buffers and switch boxes; these networks route each clock to the center of each clock sector that uses it. This highly flexible network can be used to implement a conventional full H-tree, multiple smaller H-trees, or irregular skew-balanced trees as determined by the CAD tools to fit more clocks and minimize clock skew (Ebeling et al. 2016). This clock distribution network has many programmable switches, but unlike conventional programmable routing the switch pattern is optimized for the creation of balanced structures like H-trees and the wires are designed for low process variation and are shielded against crosstalk.

The Ultrascale+ architecture from Xilinx implements clock enable circuitry for power reduction and programmable delay chains for time borrowing at the spine level as illustrated in Fig. 14b. This causes a less than 1% increase in the FPGA die size. When combined with pulse latches, these additional programmable delay chains can increase clock frequency by 5–8%, depending on the available hold margin (Ganusov and Devlin 2016). The Versal architecture leverages these programmable delay chains further by calibrating them on power up to account for process variations across the chip (Gaide et al. 2019). This *adaptive deskewing* technique helps reduce the clock uncertainty and allows the chip to run faster by having narrower guard bands in the timing models.

On-chip Memory

FFs in logic blocks were the first storage elements to be integrated into FPGAs, as described in the "Programmable Logic Blocks" section. However, as FPGA logic capacity grew, they were used to implement more complex systems which almost

always require memory to buffer and re-use data. This motivated more on-chip storage options, since building large RAMs out of registers and LUTs is over 100× less dense than a dedicated SRAM memory array. At the same time, the memory requirements of applications implemented on FPGAs are very diverse, including (but not limited to) small coefficient storage RAMs for FIR filters, large buffers for network packets, caches and register files for processor-like modules, read-only memory for instructions, and FIFOs of myriad sizes to decouple computation modules. This means that there is no single RAM configuration (capacity, word width, number of ports) that can satisfy the needs of all FPGA designs, making it challenging to decide on what kind(s) of RAM blocks should be added to an FPGA such that they are efficient for a broad range of uses. The first FPGA to include hard functional blocks for memory (*block RAMs* or *BRAMs*) was the Altera Flex 10 K in 1995. It included columns of small (2 Kb) BRAMs that connect to the rest of the fabric through the programmable routing. Since then, the capacity and diversity of FPGA on-chip memories have been gradually increasing and it is typical for ∼25% of the area of a modern FPGA to be consumed by BRAM tiles (including their programmable routing) (Tatsumura et al. 2016).

Figure 15 illustrates the organization of an SRAM-based BRAM. An FPGA BRAM consists of a traditional SRAM memory array at its core, with additional *peripheral circuitry* that makes them configurable for different purposes and provides flexible connectivity to the programmable routing. The core memory array consists of a two-dimensional array of SRAM cells to store bits, and a considerable amount of peripheral circuitry to orchestrate access to these cells for read/write operations. To simplify timing of the read and write operations, all modern FPGA BRAMs register all their inputs; they also include output registers, but these are configurable and can be bypassed. During a write operation, the column decoder activates the write drivers (WD), which in turn charge the bitlines (BL and $\overline{BL}$) according to the input data to-be-written to the memory cells. Simultaneously, the row decoder activates the wordline (WL) of the row specified by the input write address, connecting one row of cells to their bitlines so they are overwritten with new data. During a read operation, both the BL and $\overline{BL}$ are pre-charged high and then the row decoder activates the wordline of the row specified by the input read address. The contents of the activated cells cause a slight difference in the voltage between BL and $\overline{BL}$, which is sensed and amplified by the sense amplifier (SA) circuit to produce the output data (Tatsumura et al. 2016).

BRAM capacity, data word width, and number of read/write ports are all key architectural parameters. More capable BRAMs cost more silicon area, so architects must carefully balance BRAM design choices while taking into account the most common use cases in application circuits. For example, the area occupied by the memory cells grows linearly with the capacity of the BRAM, but the area of the peripheral circuitry and the number of routing ports grows sub-linearly. This means that larger BRAMs have lower area per bit, making large on-chip buffers more efficient. On the other hand, if an application requires only small RAMs, much of the capacity of a larger BRAM may be left unused. Similarly, a BRAM with a

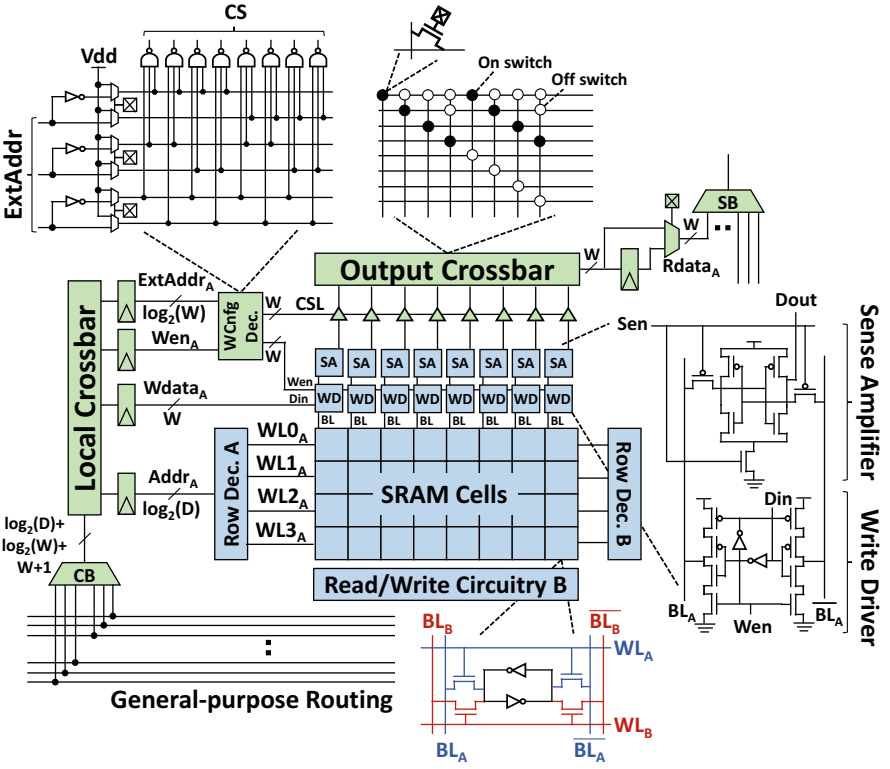


Fig. 15 Organization and circuitry of a conventional dual-port SRAM-based FPGA BRAM. The components highlighted in blue are common in any SRAM-based memory module, while those highlighted in green are FPGA-specific. This BRAM has a maximum data width of 8 bits, but the output crossbar is configured for 4-bit output mode

larger data width can provide higher data bandwidth to downstream logic. However, it costs more area than a BRAM with the same capacity but a smaller word width, as the larger data word width necessitates more sense amplifiers, write drivers and programmable routing ports. Finally, increasing the number of read/write ports to a BRAM increases the area of both the SRAM cells and the peripheral circuitry, but again increases the data bandwidth the BRAM can provide and allows more diverse uses. For example, FIFOs (which are ubiquitous in FPGA designs) require both a read and a write port. The implementation details of a dual-port SRAM cell is shown at the bottom of Fig. 15. Implementing a second port to the SRAM cell (port B highlighted in red) adds two transistors, increasing the area of the SRAM cells by 33%. In addition, the second port also needs an additional copy of the sense amplifiers, write drivers and row decoders (the "Read/Write Circuitry B" and "Row Decoder B" blocks in Fig. 15). If both ports are read/write (r/w), we also have to double the number of ports to the programmable routing.

Because the FPGA on-chip memory must satisfy the needs of *every* application implemented on that FPGA, it is also common to add extra configurability to BRAMs to allow them to adapt to application needs (Wilton et al. 1995). FPGA BRAMs are designed to have configurable width and depth by adding low-cost multiplexing circuitry to the peripherals of the memory array. For example, in Fig. 15 the actual SRAM array is implemented as a 4×8-bit array, meaning it naturally stores 8-bit data words. By adding multiplexers controlled by 3 address bits to the output crossbar, and extra decoding and enabling logic to the read/write circuitry, this RAM can also operate in 8×4-bit, 16×2-bit or 32×1-bit modes. The multipliexers in the width configurability decoder("WCnfg Dec." in Fig. 15) select between V_{dd} and address bits to implement configurable width of between 1 and 8 bits per word for example. The multiplexers are programmed using configuration SRAM cells and are used to generate column select (CS) and write enable (Wen) signals that control the sense amplifiers and write drivers for narrow read and write operations, respectively. For typical BRAM sizes (several Kb or more), the cost of this additional width configurability circuitry is small compared to the cost of a conventional SRAM array and it does not require any additional costly routing ports.

Another unique component of the FPGA BRAMs compared to conventional memory blocks is their interface to the programmable routing fabric. This interface is generally designed to be similar to that of the logic blocks described in the "Programmable Logic Blocks" section; it is easier to create a routing architecture that balances flexibility and cost well if all block types connect to it in similar ways. Connection block multiplexers, followed by local crossbars in some FPGAs, form the BRAM input routing ports, while the read outputs drive switch block multiplexers to form the output routing ports. These routing interfaces are costly, particularly for small BRAMs; they constitute 5% of the area of 256 Kb BRAM tiles, and this portion grows to 35% for smaller 8 Kb BRAMs (Yazdanshenas et al. 2017). This motivates minimizing the number of routing ports to a BRAM as much as possible without unduly comprising its functionality. Table 2 summarizes the number of routing ports required for different numbers and types of BRAM read and write ports. For example, a single-port BRAM (1r/w) requires $W + \log_2(D)$ input ports for write data and read/write address, and W output ports for read data, where W and D are the maximum word width and the BRAM depth, respectively. The table shows that a true dual-port (2r/w) BRAM requires $2W$ more ports compared to a simple dual-port (1r+1w) BRAM, which significantly increases the cost of the routing interfaces. While true dual-port memory is useful for register files, caches

Table 2 Number of routing ports needed for different numbers and types of BRAM read/write ports (W: data width, D: BRAM depth)

BRAM ports	BRAM mode	# Routing ports
1r	Single-port ROM	$\log_2(D) + W$
1r/w	Single-port RAM	$\log_2(D) + 2W$
1r+1w	Simple dual-port RAM	$2\log_2(D) + 2W$
2r/w	True dual-port RAM	$2\log_2(D) + 4W$
2r+2w	Quad-port RAM	$4\log_2(D) + 4W$

and shared memory switches, the most common use of multi-ported RAMs on FPGAs is for FIFOs, which require only one read and one write port (1r+1w rather than 2r/w ports). Consequently, FPGA BRAMs typically have true dual-port SRAM cores but with only enough routing interfaces for simple-dual port mode at the full width supported by the SRAM core (W), and limit the width of the true-dual port mode to only half of the maximum width ($W/2$).

Another way to mitigate the cost of additional BRAM ports is to *multi-pump* the memory blocks by operating the BRAMs at a frequency that is a multiple of that used for the rest of the design logic. By doing so, a physically single-ported SRAM array can implement a logically multi-ported BRAM without the cost of additional ports as in Tabula's *Spacetime* architecture (Halfhill 2010). Multi-pumping can also be used with conventional FPGA BRAMs by building the time-multiplexing logic in the soft fabric (LaForest et al. 2012); however, this leads to aggressive timing constraints for the time-multiplexing logic, which can make timing closure more challenging and increase compile time. For example, Ahmed et al. (2019) showed that careful design partitioning, floorplanning and iterative compilation are necessary for meeting timing on the time-multiplexing logic especially when using a large number of multi-pumped BRAMs. Altera introduced quad-port BRAMs in its Mercury devices in the early 2000s to make shared memory switches (useful in packet processing) and register files more efficent. However, this feature increased the BRAM size and was not sufficiently used to justify its inclusion in subsequent FPGA generations. Instead designers use a variety of techniques to combine dual-ported FPGA BRAMs and soft logic to make highly-ported structures when needed, albeit at lower efficiency (LaForest et al. 2012). We refer the interested reader to both Tatsumura et al. (2016) and Yazdanshenas et al. (2017) for extensive details about the design of BRAM core and peripheral circuitry.

In addtition to BRAMs, most FPGAs can re-use at least some of their LUTs as memory. The truth tables in the logic block K-LUTs are $2^K \times 1$-bit read-only memories; they are written once by the configuration circuitry when the design bitstream is loaded. Since LUTs already have read circuitry (read out a stored value based on a K-bit input/address), they can be used as small LUT-based RAMs (LUT-RAMs) just by adding low-cost designer-controlled write circuitry. However, a major concern is the number of additional routing ports necessary to implement the write functionality to change a LUT to a LUT-RAM. For example, an ALM in recent Altera/Intel architectures is a 6-LUT that can be fractured into two 5-LUTs and has 8 input routing ports, as explained in the "Programmable Logic Blocks" section. This means it can operate as a 64×1-bit or a 32×2-bit memory with 6 or 5 bits for read address, respectively. This leaves only 2 or 3 unused routing ports, which are not enough for write address, data, and write enable (8 total signals) if we want to read and write in each cycle (simple dual-port mode), which is the most commonly used RAM mode in FPGA designs. To overcome this problem, an entire logic block of 10 ALMs is configured as a LUT-RAM to amortize the control circuitry and address bits across 10 ALMs. The write address and write enable signals are assembled by stealing a single unused routing port

from each ALM and broadcasting the resulting address and enable to all the ALMs in a logic block (Lewis et al. 2009). Consequently, a logic block can implement a 64×10-bit or 32×20-bit simple dual-port RAM, but has a restriction that a single logic block cannot mix logic and LUT-RAM. Xilinx Ultrascale similarly converts an entire logic block to LUT-RAM, but all the routing ports of one out of the eight LUTs in a logic block are repurposed to drive the shared write address and enable signals. Therefore, a Xilinx logic block can implement a 64×7-bit or 32×14-bit simple dual-port RAM, or a slightly wider single-port RAM (64×8-bit or 32×16-bit). Avoiding extra routing ports keeps the cost of LUT-RAM low, but it still adds some area. Since it would be very unusual for a design to use more than 50% of the logic fabric as LUT-RAMs, both Altera/Intel and Xilinx have elected to make only half (or less) of their logic blocks LUT-RAM capable in their recent architectures, thereby further reducing the area cost.

Designers require many different RAMs in a typical design, all of which must be implemented by the fixed BRAM and LUT-RAM resources on the chip. Forcing designers to determine the best way to combine BRAM and LUT-RAM for each memory configuration they need and writing Verilog to implement them would be laborious and would also impede migration of the design to a new FPGA architecture. Instead, the vendor CAD tools include a *RAM mapping* stage that implements the *logical* memories in the user's design using the *physical* BRAMs and LUT-RAMs on the chip. The RAM mapper chooses the physical memory implementation (i.e., memory type and the width/number/type of its ports) and generates any additional logic required to combine multiple BRAMs or LUT-RAMs to implement each logical RAM. An example of mapping a logical 2048×32-bit RAM with 2 read and 1 write ports to an FPGA with physical 1024×8-bit dual-port BRAMs is illustrated in Fig. 16. First, four physical BRAMs are combined in parallel to make wider RAMs with no extra logic. Then, soft logic resources are used to perform depth-wise stitching of two sets of four physical BRAMs, such that

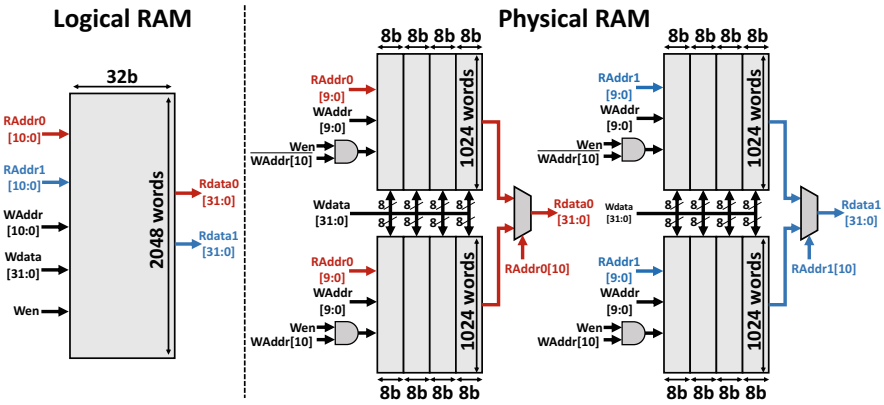


Fig. 16 Mapping a 2048×32-bit 2r+1w logical RAM to an FPGA with 1024×8-bit 1r+1w physical BRAMs

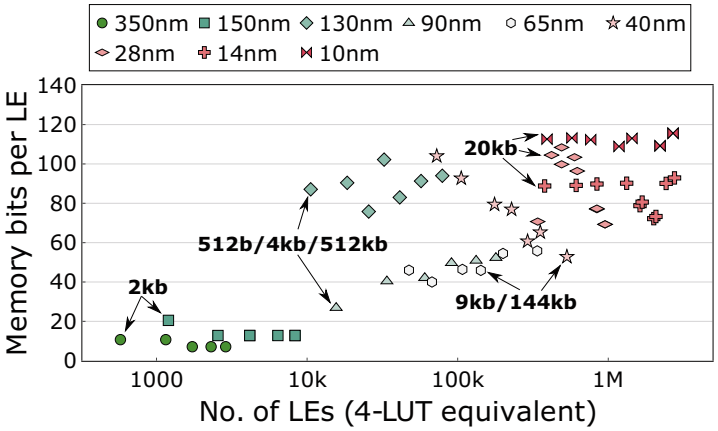


Fig. 17 Memory bits per logic elements for different generations of Altera/Intel FPGAs starting from the 350 nm Flex 10K (1995) to the 10 nm Agilex (2019) architecture. FPGA on-chip memory density has increased by a factor of 16× in the last 25 years. The labels show the sizes of BRAMs in each generation

the most-significant bits of the write and read addresses are used as write enable and read output multiplexer select signals, respectively. Finally, in this case, we require two read ports and one write port while the physical BRAMs only support a maximum of 2r/w ports. To implement the second read port, the whole structure is either replicated as shown in the figure or double-pumped as previously explained. Several algorithms for optimizing RAM mapping are described in Tessier et al. (2007) and Lai and Lin (2016).

Over the past 25 years, FPGA memory architecture has evolved considerably and has also become increasingly important, as the ratio of memory to logic on an FPGA die has grown significantly. Figure 17 plots the memory bits per logic element (including LUT-RAM) versus the number of logic elements in Altera/Intel devices starting from the 350 nm Flex 10K devices (1995) to 10 nm Agilex devices (2019). There has been a gradual increase in the memory richness of FPGAs over time, and to meet the demand for more bits at a cheaper cost, modern BRAMs have larger capacities (20 Kb) than the first BRAMs (2 Kb). Some FPGAs have had highly heterogeneous BRAM architectures in order to provide some physical RAMs that are efficient for small or wide logical RAMs, and others that are efficient for large and relatively narrow logical RAMs. For example, Stratix (130 nm) had 3 types of BRAM, with capacities of 512 b, 4 Kb and 512 Kb. The introduction of LUT-RAM in Stratix III (65 nm) reduced the need for small BRAMs, so it moved to a memory architecture with only medium and large size (9 Kb and 144 Kb) BRAMs. Stratix V (28 nm) and later Intel devices have moved to a combination of LUT-RAM and a single medium-sized BRAM (20 Kb) to simplify both the FPGA layout as well as RAM mapping and placement. A similar trend can be observed in Xilinx devices (Tatsumura et al. 2016); Xilinx's RAM architecture also combines LUT-RAM and a

medium-sized 18 Kb RAM, but also includes hard circuitry to combine two BRAMs into a single 36 Kb block. However, Xilinx's most recent devices add a large 288 Kb BRAM (UltraRAM) to be more efficient for very large buffers, showing that there is still no general agreement on the best BRAM architecture. Some recent Intel devices further enhance their memory capacity by integrating the FPGA fabric with one or more embedded SRAM (eSRAM) chiplets using interposer technology that will be discussed in the "Interposers" section later. Each eSRAM chiplet implements eight large simple dual-port memories with a combined capacity of 47 Mb in Stratix 10 and 18 Mb in Agilex. These memories are ideal for wide and deep buffers that exceed on-chip storage capacity, but benefit from reduced latency; for example, routing tables or packet headers in networking applications.

To give some insight into the relative areas and efficiencies of different BRAMs, Table 3 shows the resource usage, silicon area, and frequency of a 2048×72-bit logical RAM when it is implemented by Quartus (the CAD flow for Altera/Intel FPGAs) in a variety of ways on a Stratix IV device. The silicon areas are computed using the published Stratix III block areas from Wong et al. (2011) and scaling them from 65 nm down to 40 nm, as Stratix III and IV have the same architecture but use different process nodes. As this logical RAM is a perfect fit to the 144 Kb BRAM in Stratix IV, it achieves the best area when mapped to a single 144 Kb BRAM. Interestingly, mapping to eighteen 9 Kb BRAMs is only 1.9× larger in silicon area (note that output width limitations lead to 18 BRAMs instead of the 16 one might expect). The 9 Kb BRAM implementation is actually faster than the 144 Kb BRAM implementation, as the smaller BRAMs have higher maximum operating frequencies. Mapping such a large logical RAM to LUT-RAMs is inefficient, requiring 12.7× more area and running at 40% of the frequency. Finally, mapping only to the logic and routing resources highlights the importance of BRAMs; the area is over 300× larger than the 144 Kb BRAM implementation. While the 144 Kb BRAM is most efficient for this single test case, real designs have diverse logical RAMs, and for small or shallow memories the 9 Kb and LUT-RAM options would outperform the 144 Kb BRAM, motivating a diversity of on-chip RAM resources. To choose the best mix of BRAM sizes and maximum word widths, one needs both a RAM mapping tool and tools to estimate the area, speed and power of each BRAM (Yazdanshenas et al. 2017). Published studies into BRAM architecture trade-offs for FPGAs include (Yazdanshenas et al. 2017; Lewis et al. 2013).

Until now, all commercial FPGAs use only SRAM-based memory cells in their BRAMs. With the desire for more dense BRAMs that would enable more memory-

Table 3 Implementation results for a 2048×72-bit 1r+1w RAM using BRAMs, LUT-RAMs and registers on Stratix IV

Implementation	Half-ALMs	BRAMs		Area (mm^2)	Freq. (MHz)
		9K	144K		
144K BRAMs	0	0	1	0.22 (1.0×)	336 (1.0×)
9K BRAMs	0	18	0	0.41 (1.9×)	497 (1.5×)
LUT-RAM	6597	0	0	2.81 (12.8×)	134 (0.4×)
Registers	165155	0	0	68.8 (313×)	129 (0.4×)

rich FPGAs and SRAM scaling becoming increasingly difficult due to process variation, a few academic studies have explored the use of other emerging memory technologies such as magnetic tunnel junctions (MTJs) to build FPGA memory blocks. According to Tatsumura et al. (2016), MTJ-based BRAMs could increase the FPGA memory capacity by up to 2.95× with the same die size; however, they would increase the process complexity.

DSP Blocks

Initially, the only dedicated arithmetic circuits in commercial FPGA architectures were carry chains to implement efficient adders, as discussed in the "Programmable Logic Blocks" section earlier. Thus, multipliers had to be implemented in the soft logic using a combination of LUTs and carry chains, which for larger operand bit widths incurs significant logic utilization and delay. As wireless communication and signal processing became major FPGA markets, system designers proposed novel implementations to mitigate the inefficiency of multiplier implementations in soft logic. For example, the multiplier-less *distributed arithmetic* technique was proposed to implement efficient FIR filter structures in LUTs (Meher et al. 2008).

With the prevalence of multipliers in FPGA designs from key application domains and their lower efficiency when implemented in soft logic, they quickly became a candidate for hardening as dedicated circuits in FPGA architectures. An N-bit multiplier array consists of N^2 logic gates to generate partial products and compression trees to reduce them, with only $2N$ inputs and $2N$ outputs. Therefore, the high gains of hardening the multiplier logic and the relatively low cost of the programmable interfaces to the FPGA's routing fabric strongly advocated for adopting hard multipliers in subsequent FPGA architectures. As shown in the top left of Fig. 18, Xilinx introduced its Virtex-II architecture with the industry's first 18×18 bit hard multiplier blocks. To simplify the layout integration with the full-custom FPGA fabric, these multipliers were arranged in columns right beside BRAM columns. In order to further reduce the interconnect cost, the multiplier block and its adjacent BRAM had to share some interconnect resources, limiting the maximum usable data width of the BRAM block when the multiplier is used for computation. Multiple hard 18-bit multipliers could be stitched together with soft logic to form bigger multipliers or FIR filters.

In 2002, Altera adopted a different approach by introducing more fully-featured DSP blocks targeting the communications and signal processing domains in their Stratix architecture (Lewis et al. 2003) (see the second block in Fig. 18). The main design philosophy of this DSP block was to minimize the amount of soft logic resources used to implement common DSP algorithms by hardening more functionality inside the DSP block and enhancing its flexibility to allow more applications to use it. The Stratix DSP block was highly configurable with support for different modes of operation and multiplication precisions unlike the fixed-function 18-bit multipliers in Virtex-II. Each Stratix variable-precision DSP block spanned 8 FPGA rows and could implement eight 9×9 bit multipliers, four 18×18 bit multipliers, or one 36×36 multiplier.

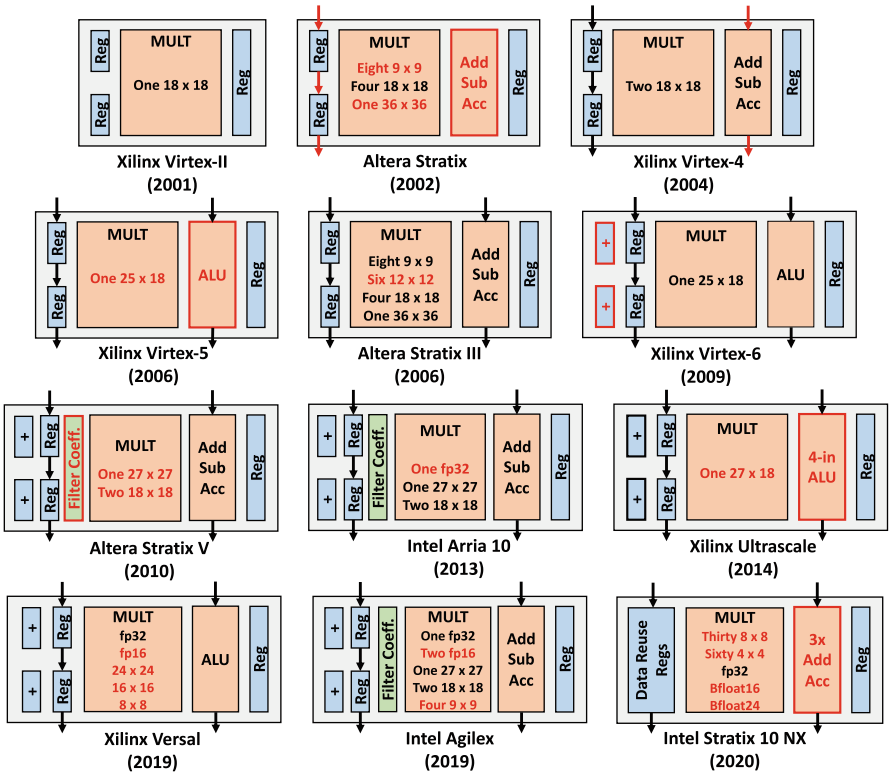


Fig. 18 DSP block evolution in Altera/Intel and Xilinx FPGAs. Incrementally added features are highlighted in red

These modes of operation selected by Altera highlight an important theme of designing FPGA hard blocks: increasing the flexibility and utility of these blocks by adding low-cost circuitry such that it becomes more broadly useful. For example, an 18×18 multiplier array can be decomposed into two 9×9 arrays that together use the same number of inputs and outputs (and hence routing ports). Similarly, four 18×18 multipliers can be combined into one 36×36 array using cheap glue logic. Figure 19 shows how an 18×18 multiplier array can be fractured into multiple 9×9 arrays. It can be split into four 9×9 arrays by doubling the number of input and output pins. However, to avoid adding these costly routing interfaces, two of the four 9×9 arrays are left unused (grey circles) and the other two (blue circles) are used to perform the two multiplications $A0 \times B0$ and $A1 \times B1$. This is done by splitting the partial product compressor trees at the positions indicated by the red dashed lines and adding inverting capabilities to the border cells of the top-right array, marked with crosses in Fig. 19 to implement two's complement signed multiplication using the Baugh-Wooley algorithm (the bottom left array already has the inverting capability from the 18×18 array).

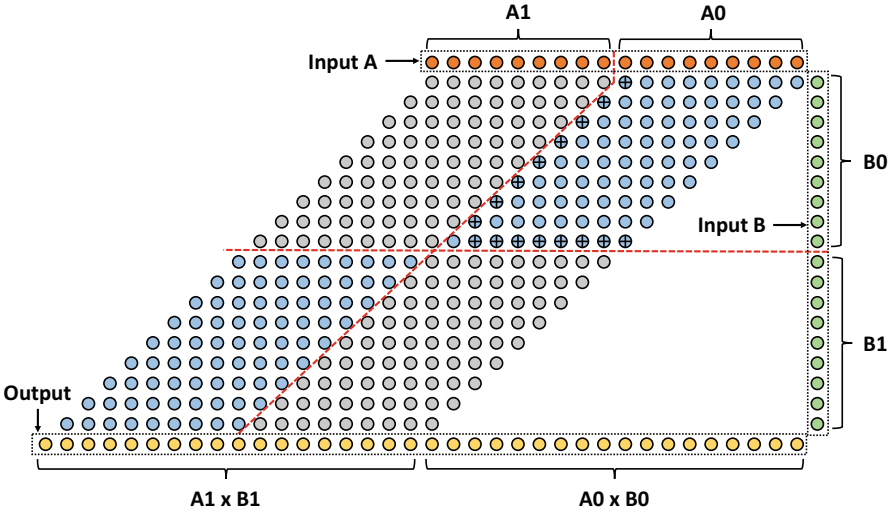


Fig. 19 Fracturing an 18×18 multiplier array into two 9×9 arrays with the same number of input/output ports

In addition to the fracturable multiplier arrays, the Stratix DSP also incorporated an adder/output block to perform summation and accumulation operations, as well as hardened input registers that could be configured as shift registers with dedicated cascade interconnect between them to implement efficient FIR filter structures. Xilinx also adopted a fully-featured DSP block approach by introducing their *DSP48* tiles in the Virtex-4 architecture. Each DSP tile had two fixed-precision 18×18 bit multipliers with similar functionalities to the Stratix DSP block (e.g., input cascades, adder/subtractor/accumulator). Virtex-4 also introduced the ability to cascade the adders/accumulators using dedicated interconnects on the output side of the DSP blocks to implement high-speed systolic FIR filters with hardened reduction chains.

An N-tap FIR filter performs a discrete 1D convolution between the samples of a signal $X = \{x_0, x_1, \ldots, x_T\}$ and certain coefficients $C = \{c_0, c_1, \ldots, c_{N-1}\}$ that represent the impulse response of the desired filter, as shown in Eq. (1).

$$y_n = c_0 x_n + c_1 x_{n-1} + \ldots + c_N x_{n-N} = \sum_{i=0}^{N} c_i x_{n-i} \tag{1}$$

Many of the FIR filters used in practice are symmetric with $c_i = c_{N-i}$, for $i = 0$ to $N/2$. As a result of this symmetry, the filter computation can be refactored as shown in Eq. (2).

$$y_n = c_0[x_n + x_{n-N}] + \ldots + c_{N/2-1}[x_{n-N/2-1} + x_{n-N/2}] \tag{2}$$

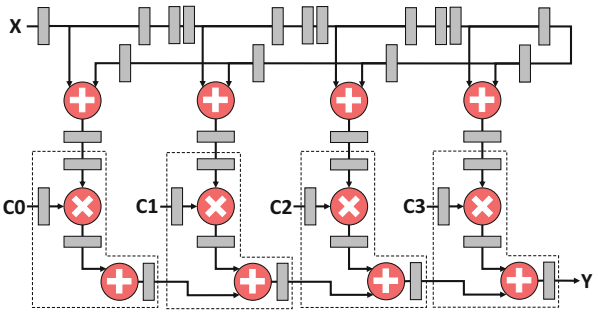


Fig. 20 Systolic implementation of a symmetric FIR filter circuit

Figure 20 shows the structure of a systolic symmetric FIR filter circuit, which is a key use case for FPGAs in wireless base stations. Both Stratix and Virtex-4 DSP blocks can implement the portions highlighted by the dotted boxes, resulting in significant efficiency gains compared to implementing them in the FPGA's soft logic. Interestingly, while FPGA CAD tools will automatically implement a multiplication operation (written as a * operator in RTL) in DSP blocks, they will generally not make use of any of the advanced DSP block features (e.g., accumulation, systolic registers for FIR filters) unless a designer manually instantiates a vendor-supplied DSP block IP in the proper mode. Consequently, using the more powerful DSP block features makes a design less portable when migrating to another FPGA with different DSP block capabilities. Some work has extended automatic DSP block inference to sequences of multiply, add and subtract operations in RTL that exactly match the DSP block capabities (Ronak and Fahmy 2015a). This can improve automatic inference to some extent, but it will be difficult to extend to fully utilize advanced DSP block features like coefficient re-use networks.

The Stratix III/IV DSP block was similar to the Stratix II one but could implement four 18×18 multipliers per half a DSP block (instead of two) if their results are summed to limit the number of output routing interfaces. Table 4 lists the implementation results of both symmetric and asymmetric 51-tap 16-bit FIR filters, with and without using the hard DSP blocks on a Stratix IV device. When DSP blocks are not used, we experiment with two different cases: fixed filter coefficients, and filter coefficients that can change at runtime. If the filter coefficients are fixed, the multiplier arrays implemented in the soft logic are optimized by synthesizing away parts of the partial product generation logic that correspond to zero bits in the coefficient values. Hence, it has lower resource utilization than with input coefficients that can change at runtime. For the symmetric filter, even when using the DSP blocks, we still need to use some soft logic resources to implement the input cascade chains and pre-adders, as shown in Fig. 20. Using the hard DSP blocks results in 3× higher area efficiency vs. using the soft fabric in the case of fixed coefficients. This gap grows to 6.2× for filter coefficients that are changeable during runtime. For the asymmetric filter, the complete FIR filter structure can be implemented in the DSP blocks without any soft logic resources. Thus, the

Table 4 Implementation results for a 51-tap 16-bit FIR filter on Stratix IV with and without using the hardened DSP blocks

Symmetric Filter				
Implementation	Half-ALMs	DSPs	Area (mm^2)	Freq. (MHz)
With DSPs	403	$3\frac{2}{8}$	0.49 (1.0×)	510 (1.0×)
Without DSPs (fixed coeff.)	3505	0	1.46 (3.0×)	248 (0.5×)
Without DSPs (variable coeff.)	7238	0	3.01 (6.2×)	220 (0.4×)
Asymmetric Filter				
Implementation	half-ALMs	DSPs	Area (mm^2)	Freq. (MHz)
With DSPs	0	$6\frac{3}{8}$	0.63 (1.0×)	510 (1.0×)
Without DSPs (fixed coeff.)	5975	0	2.48 (3.9×)	245 (0.5×)
Without DSPs (variable coeff.)	12867	0	5.35 (8.5×)	217 (0.4×)

area efficiency gap increases to 3.9× and 8.5× for fixed and variable coefficients, respectively. These gains are large but still less than the 35× gap between FPGAs and ASICs (Kuon and Rose 2007) usually cited in academia. The difference is partly due to some soft logic remaining in most application circuits, but even in the case where the FIR filter perfectly fits into DSP blocks with no soft logic, the area reduction hits a maximum of 8.5×. The primary reasons for the lower than 35× gain of Kuon and Rose (2007) are the interfaces to the programmable routing and the general inter-tile programmable routing wires and muxes that must be implemented in the DSP tile. In all cases, using the hard DSP blocks results in about 2× frequency improvement as shown in Table 4. Similarly to BRAMs, the high operating frequencies of DSP blocks mean they can often be multi-pumped (run at a multiple of the soft logic frequency); this is mainly used for resource reduction in DSP-bound designs as in Ronak and Fahmy (2015b).

The next few FPGA architecture generations from both Altera and Xilinx witnessed only minor changes in the DSP block architecture. The main focus of both vendors was to fine-tune the DSP block capabilities for emerging application domains without adding costly programmable routing interfaces. In Stratix V, the DSP block was greatly simplified to natively support two 18×18 bit multiplications (suitable for signal processing) or one 27×27 multiplication (suitable for single-precision floating-point mantissa multiplication). As a result, the simpler Stratix V DSP block spanned a single row, which is more friendly to Altera's row redundancy scheme (i.e., the ability to skip single FPGA rows with fabrication faults in them to increase the effective yield). In addition, input pre-adders as well as embedded coefficient banks to store read-only filter weights were added, which allowed implementation of the whole symmetric FIR filter structure shown in Fig. 20 inside the DSP blocks without the need for any soft logic resources. Xilinx followed a similar path in incorporating 27×18 multiplication with support for pre-adders in Virtex-6 DSP blocks.

As shown in Fig. 18, Xilinx DSP blocks since Virtex-5 have incorporated an ALU that can perform logic operations as well as add and subtract; both the ALU operation and the data paths through the DSP are selected by additional inputs so they can change dynamically from cycle to cycle. This enhancement makes these DSP blocks well suited for the datapath of a soft processor (Cheah et al. 2014). Controlling DSP operations dynamically in this manner increases the flexibility of the block, but has some area cost as adding routing input ports for dynamic control signals is more expensive than adding configuration SRAM cells to statically select operations.

As illustrated in Fig. 18, up to 2009 the evolution of the DSP block architecture was mainly driven by the precisions and requirements of communication applications, especially in wireless base stations, with very few academic research explorations. With the large-scale deployment of FPGAs in datacenters and the emergence of DL as a key component of many applications both in datacenter and edge workloads, the DSP block architecture has evolved in two different directions. The first direction targets the high-performance computing (HPC) domain by adding native support for single-precision floating-point (`fp32`) multiplication. Before that, FPGA vendors would supply designers with IP cores that implement floating-point arithmetic out of fixed-point DSPs and a considerable amount of soft logic resources. This created a major barrier for FPGAs to compete with CPUs and GPUs (which have dedicated floating-point units) in the HPC domain. Native floating-point capabilities were first introduced in Intel's Arria 10 architecture, with a key design goal of avoiding a large increase in DSP block area (Langhammer and Pasca 2015). By reusing the same interface to the programmable routing, not supporting uncommon features like subnormals, flags and multiple rounding schemes, and maximizing the reuse of existing fixed-point hardware, the block area increase was limited to only 10% (which translates to 0.5% total die area increase). Floating-point capabilities are supported in all subsequent generations of Intel FPGAs and in the *DSP58* tiles of the Xilinx Versal architecture (Gaide et al. 2019).

The second direction targets increasing the density of low-precision integer multiplication specifically for DL inference workloads. Prior work has demonstrated the use of low-precision fixed-point arithmetic (8-bit and below) instead of `fp32` at negligible or no accuracy degradation, but greatly reduced hardware cost (Wang et al. 2019). However, the required precision is model-dependent and can even vary between different layers of the same model. As a result, FPGAs have emerged as an attractive solution for DL inference due to their ability to implement custom precision datapaths. This has led both academic researchers and FPGA vendors to investigate adding native support for low-precision multiplication to DSP blocks. Boutros et al. (2018) enhanced the fracturability of an Intel-like DSP block to support more `int9` and `int4` multiply and MAC operations, while keeping the same DSP block routing interface and ensuring its backward compatibility. The proposed DSP block could implement four `int9` and eight `int4` multiply/MAC operations along with Arria-10-like DSP block functionality at the cost of 12% DSP block area increase, which is equivalent to only 0.6% increase in total die area. This DSP block increased the performance of 8-bit and 4-bit DL accelerators

by 1.3× and 1.6× while reducing the utilized FPGA resources by 15% and 30% respectively, compared to an FPGA with DSPs that do not natively support these modes of operation. Another academic work (Rasoulinezhad et al. 2019) enhanced a Xilinx-like DSP block by including a fracturable multiplier array instead of the fixed-precision multiplier in the *DSP48E2* block to support `int9`, `int4` and `int2` precisions. It also added a FIFO register file and special dedicated interconnect between DSP blocks to enable more efficient standard, point-wise and depth-wise convolution layers. Shortly after, the Intel Agilex DSP block added support for an `int9` mode of operation along with half-precision floating-point (`fp16`) and brain float (`bfloat16`) precisions as well. Also, the Xilinx Versal architecture now natively supports `int8` multiplications in its *DSP58* tiles (Gaide et al. 2019).

Throughout the years, the DSP block architecture has evolved to best suit the requirements of key application domains of FPGAs, and provide higher flexibility such that many different applications can benefit from its capabilities. The common focus across all the steps of this evolution was reusing multiplier arrays and routing ports as much as possible to best utilize both these costly resources. However, this becomes harder with the recent divergence in the DSP block requirements of key FPGA application domains between high-precision floating-point in HPC, medium-precision fixed-point in communications, and low-precision fixed-point in DL. As a result, Intel introduced its first domain-specialized FPGA optimized for artificial intelligence (AI) workloads, the Stratix 10 NX. This new FPGA replaces conventional DSP blocks with AI tensor blocks (Langhammer et al. 2021). The tensor blocks drop the support for legacy DSP modes and precisions that were targeting the communications domain and adopt new ones targeting the DL domain specifically. This tensor block significantly increases the number of `int8` and `int4` MACs to 30 and 60 per block respectively, at almost the same die size. Feeding all multipliers with inputs without adding more routing ports is a key concern. Accordingly, the NX tensor block introduces a double-buffered data reuse register network that can be sequentially loaded from a smaller number of routing ports, while allowing common DL compute patterns to make the best use of all available multipliers. Recent work has shown that the Stratix 10 NX with tensor blocks can deliver an average 3.5× performance boost compared to FPGAs with conventional DSP blocks for real-time DL inference workloads (Boutros et al. 2020).

Processor Subsystems

As the complexity of FPGA applications increased, many designs required a software-programmable processor for lightweight control, housekeeping, performance monitoring, or debugging. Therefore, designers had to build and optimize their own *soft processors* out of the FPGA's programmable function blocks and routing (as shown in Fig. 21a), which was a significantly laborious and challenging task. To facilitate the integration of processor subsystems in FPGA designs, FPGA vendors supplied heavily optimized and parameterized soft processor IPs that designers can readily use such as the Nios and Microblaze soft processors from

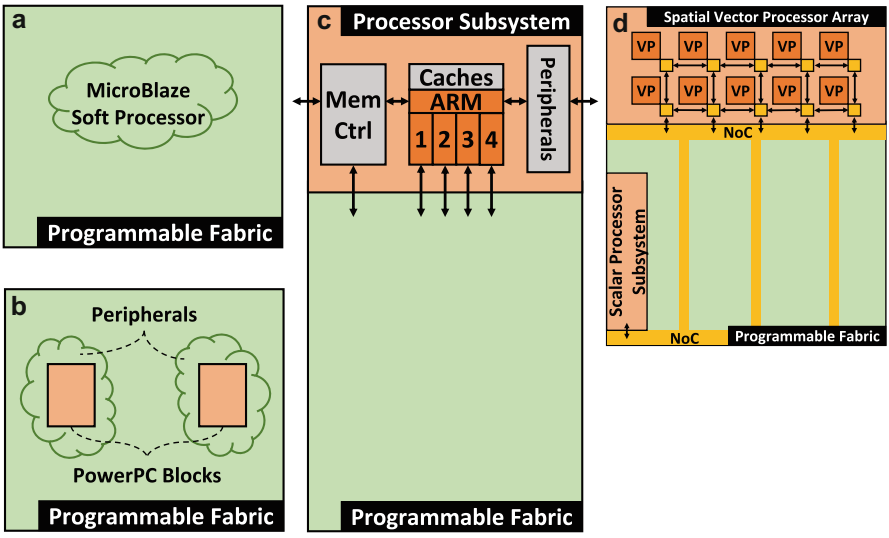


Fig. 21 (**a**) Early Xilinx FPGA with a MicroBlaze soft processor implemented in soft logic, (**b**) Xilinx Virtex-II Pro FPGA with 2 hard PowerPC blocks whose peripherals are implemented in soft logic, (**c**) Xilinx Zynq Ultrascale+ with a complete hard processor subsystem, and (**d**) Xilinx Versal architecture with both a hard scalar processor subsystem and a spatial vector processor array

Altera and Xilinx, respectively. This alleviated the design burden from FPGA users while still allowing them to flexibly configure their architecture parameters (e.g., instruction/data cache sizes, number of cache levels, ALU capabilities, etc.) to match the application requirements. However, these soft processors are still area-inefficient, slower, and have limited capabilities (e.g., scalar, single-issue, in-order microarchitecture) compared to mainstream CPUs, which makes them more suitable for lightweight control and housekeeping tasks rather than compute-oriented ones. The gap is even larger compared to direct hardware execution on repetitive tasks. For example, a Nios II soft processor on a Stratix IV FPGA runs at 250 MHz and consumes 1130 LUTs, 4 DSPs, and 11 BRAMs. When used to compute a simple third-degree polynomial, it has 50× less performance, 130× higher energy, and 2× higher LUT utilization compared to a dedicated hardware implementation (configured into the FPGA) of the same function. Some studies attempt to optimize scalar soft processors for more compute-intensive tasks by adding support for vector instructions. Yiannacouras et al. show that a vector soft processor can improve performance by 25× over a scalar soft processor; while area increases, the area-delay product is still 3× better than a scalar soft processor (Yiannacouras et al. 2009).

As more systems incorporated processors for control and less compute-intensive tasks, FPGA vendors began to harden processor cores to increase performance vs. soft processors. For example, the Xilinx Virtex-II Pro architecture had up to 2 IBM PowerPC RISC processor blocks as illustrated in Fig. 21b, while Altera

integrated an ARM core in the Apex architecture. These initial efforts hardened only the *raw* processor core with primitive wire interfaces to the programmable fabric, while the rest of the processor subsystem (e.g., memory controller and peripherals) had to be implemented in the soft logic. This was still time-consuming and did not show enough efficiency gains compared to soft processors to justify the higher design effort and reduced configurability; consequently, these hardened processor-core-only systems were not very successful. With FPGAs growing into more complex and heterogeneous platforms, complete hard processor subsystems (i.e., processors along with their key peripherals) have been incorporated in recent FPGA architectures. This approach has been much more successful as it provides designers with an easy-to-use software environment for implementing portions of their applications, while still achieving a significantly higher performance and energy efficiency compared to soft processors. Consequently, high-performance full-featured hard processor subsystems are now available in most FPGA families. For example, Xilinx's Zynq Ultrascale+ (in Fig. 21c) has an embedded quad-core ARM Cortex-A53 processor along with a cache coherency unit, a memory management unit, direct memory access controller, and many different IO peripherals (e.g., USB, I^2C, UARTs, GPIOs, etc.) to communicate with the outside world, as well as the tightly coupled FPGA fabric. These hybrid devices can be used in many applications where the processor handles strictly serial and branching portions of the workload while the highly-parallel compute-intensive portions are offloaded to the FPGA – this echoes the initial vision for reconfigurable computer architectures in the 1960s (Estrin 1960).

The Xilinx Versal architecture integrates not only an FPGA fabric and a traditional hard processor subsystem, but also a many-core vector processor complex with bus-based reconfigurable interconnect, as shown in Fig. 21d. This architecture still has a spatial nature (similar to an FPGA), and combines the software-level programmability of vector processors with the flexibility of programmable interconnects, making processor cores essentially another form of logic blocks in reconfigurable devices. This new architecture is initially targeted at 5G signal processing and AI, two large and compute-intensive markets for FPGAs. New tools for architecture exploration and evaluation of these highly heterogeneous devices are also emerging (Boutros et al. 2022), enabling new research into both their programming models and efficiency in various applications.

System-Level Interconnect: Network-on-Chip

As FPGAs have grown in both capacity and IO speed, distributing ever higher bandwidth data streams throughout an ever larger fabric has become challenging. Traditionally the *system-level interconnect* that connects high-speed IO interfaces such as DDR, PCIe and Ethernet to modules implemented in the FPGA fabric has been implemented as *soft* buses. These soft buses include multiplexing, arbitration, pipelining and wiring between the relevant endpoints. As the data bandwidth of external IO interfaces has increased, these soft buses have been forced to become

very wide to carry the larger data streams, increasing their resource utilization and making timing closure harder. For example, a single channel of high-bandwidth memory (HBM) has a 128-bit double data rate interface operating at 1 GHz, so a bandwidth-matched soft bus running at 250 MHz must be 1024 bits wide. With recent FPGAs incorporating up to 8 HBM channels as well as numerous PCIe, Ethernet and other interfaces, system level interconnect can rapidly use a major fraction of the FPGA logic and routing resources. In addition, system-level interconnect tends to span long distances. The combination of very wide and physically long buses makes timing closure challenging and usually requires deep pipelining of the soft bus, further increasing its resource use. The system-level interconnect challenge is becoming more difficult in advanced process nodes, as the number and speed of FPGA external interfaces increases, and the metal wire parasitics (and thus interconnect delay) scales poorly (Bohr 1995).

Abdelfattah and Betz (2013) proposed embedding a hard, packet-switched network-on-chip (NoC) in the FPGA fabric to enable more efficient and easier-to-use system-level interconnect. Although a full-featured packet-switched NoC could be implemented using the soft logic and routing of an FPGA, an NoC with hardened routers and links is 23× more area efficient, 6× faster, and consumes 11× less power compared to a *soft* NoC. Designing a hard NoC for an FPGA is challenging since the FPGA architect must commit many choices to silicon (e.g., number of routers, link width, NoC topology) yet still maintain the flexibility of an FPGA to implement a wide variety of applications using many different external interfaces and communication endpoints. Work in Abdelfattah and Betz (2013) advocates for a mesh topology with a moderate number of routers (e.g., 16) and fairly wide (128-bit) links; these choices keep the area cost to less than 2% of the FPGA while ensuring the NoC is easier to lay out and a single NoC link can carry the entire bandwidth of a DDR channel. A hard NoC must also be able to flexibly connect to user logic implemented in the FPGA fabric. Abdelfattah et al. (2015) introduced the *fabric port* which interfaces the hard NoC routers to the FPGA programmable fabric by performing width adaptation, clock domain crossing and voltage translation. This decouples the NoC from the FPGA fabric such that the NoC can run at a fixed (high) frequency, and still interface to FPGA logic and IO interfaces of different speeds and bandwidth requirements with very little glue logic. Hard NoCs also appear very well suited to FPGAs in datacenters. Datacenter FPGAs are normally configured in two parts: a *shell* provides system-level interconnect to the external interfaces, and a *role* implements the application acceleration functionality (Caulfield et al. 2016). The resource use of the shell can be significant: it requires 23% of the device resources in the first generation of Microsoft's Catapult systems (Putnam et al. 2014). Yazdanshenas and Betz (2018) showed that a hard NoC significantly improves resource utilization, operating frequency and routing congestion in datacenter FPGAs. Other studies have proposed FPGA-specific optimizations to increase the area efficiency and performance of soft NoCs (Kapre and Gray 2017; Papamichael and Hoe 2012). However, Yazdanshenas and Betz (2018) showed that even optimized soft NoCs still trail hard NoCs in usable bandwidth, latency, area and routing congestion.

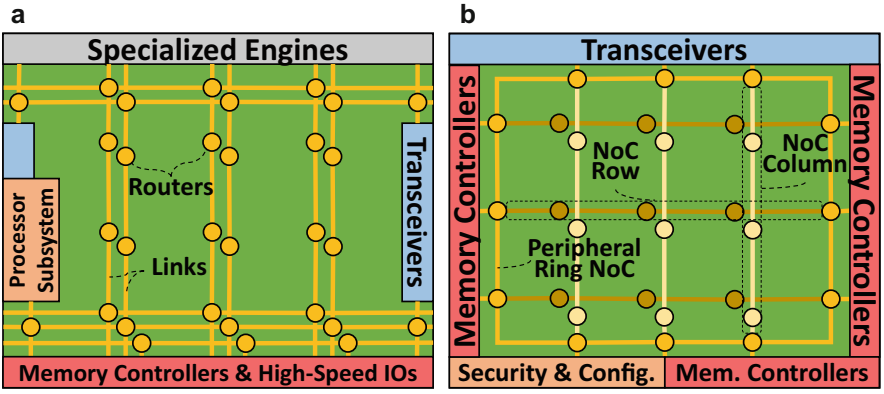


Fig. 22 Network-on-Chip system-level interconnect in next-generation (**a**) Xilinx Versal and (**b**) Achronix Speedster7t architectures

Recent Xilinx Versal and Achronix Speedster7t FPGAs integrate a hard NoC similar to the academic proposals discussed above. Versal uses a hard NoC for system-level communication between various endpoints (Gigabit transceivers, processor, AI subsystems, soft fabric), and is in fact the only way for external memory interfaces to communicate with the rest of the device (Swarbrick et al. 2019). It uses 128-bit wide links running at 1 GHz, matching a DDR channel's bandwidth. Its topology is related to a mesh, but with all horizontal links pushed to the top and bottom of the device to make it easier to lay out within the FPGA floorplan. The Versal NoC contains multiple rows (i.e., chains of links and routers) at the top and bottom of the device, and a number of vertical NoC columns (similar to any other hard block columns such as DSPs) depending on the device size as shown in Fig. 22a. The NoC has programmable routing tables that are configured at boot time and provides standard AXI interfaces as its fabric ports. The Speedster7t NoC topology is optimized for external interface to fabric transfers. It consists of a peripheral ring around the fabric with NoC rows and columns at regular intervals over the FPGA fabric as shown in Fig. 22b. The peripheral ring NoC can operate independently without configuring the FPGA fabric to route the traffic between different external interfaces. There is no direct connectivity between the NoC rows and columns; the packets from a master block connecting to a NoC row will pass through the peripheral ring to reach a slave block connected to a NoC column.

Interposers

FPGAs have been early adopters of interposer technology that allows dense interconnection of multiple silicon dice. As shown in Fig. 23a, a passive interposer is a silicon die (often in a trailing process technology to reduce cost) with conventional metal layers forming routing tracks and thousands of microbumps on its surface

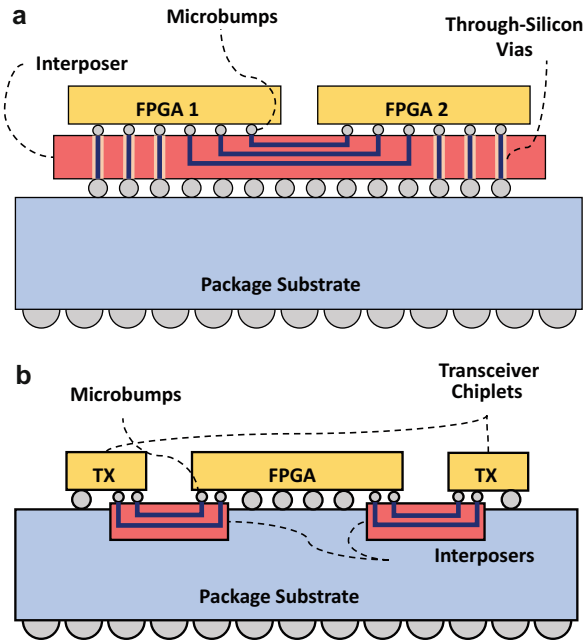


Fig. 23 Different interposer technologies used for integrating multiple chips in one package in: (**a**) Xilinx multi-die interposer-based FPGAs and (**b**) Intel devices with EMIB-connected transceiver chiplets

that connect to two or more dice flipped on top of it. One motivation for interposer-based FPGAs is achieving higher logic capacity at a reasonable cost. Both high-end systems and emulation platforms to validate ASIC designs before fabrication demand FPGAs with high logic capacity. However, large monolithic (i.e., single-silicon-die) devices have poor yield, especially early in the lifetime of a process technology (exactly when the FPGA is state-of-the-art). Combining multiple smaller dice on a silicon interposer is an alternative approach that can have higher yield. A second motivation for 2.5D systems is to enable integration of different specialized *chiplets* (possibly using different process technologies) into a single system. This approach is also attractive for FPGAs as the fabric's programmability can bridge disparate chiplet functionality and interface protocols.

Xilinx's largest devices starting from the Virtex-7 (28 nm) generation use passive silicon interposers to integrate three or four FPGA dice that each form a portion of the FPGA's rows. The largest interposer-based devices provide more than twice the logic elements of the largest monolithic FPGAs at the same process node. The FPGA programmable routing requires a large amount of interconnect, raising the question of whether the interposer microbumps (which are much larger and slower than conventional routing tracks) will limit the routability of the system. For example, in Virtex-7 interposer-based FPGAs, only 23% of the vertical routing tracks cross between dice through the interposer (Nasiri et al. 2015), with an estimated additional delay of ∼1 ns (Chaware et al. 2012). The study in Nasiri et al. (2015) showed that CAD tools that place the FPGA logic to minimize crossing of

an interposer boundary combined with architecture changes that increase the switch flexibility to the interposer-crossing tracks can largely mitigate the impact of this reduced signal count. The entire vertical bandwidth of the NoC in the Xilinx Versal architecture (discussed in the "System-Level Interconnect: Network-on-Chip" section) crosses between dice, helping to provide more interconnect bandwidth. An embedded NoC makes good use of the limited number of wires that can cross an interposer, as it runs its links at a high frequency and they can be shared by different communication streams as they are packet-switched. Xilinx has also used their interposer technology for heterogeneous integration by incorporating HBM, starting with their 16 nm Virtex Ultrascale+ generation.

Intel FPGAs instead use smaller interposers called embedded multi-die interconnect bridges (EMIB) carved into the package substrate as shown in Fig. 23b. Intel Stratix 10 devices use EMIB to integrate a large FPGA fabric die with smaller IO transceiver or HBM chiplets in the same package, decoupling the design and process technology choices of these two crucial elements of an FPGA. Some recent studies (Nurvitadhi et al. 2018, 2019) used EMIB technology to tightly couple an FPGA fabric with specialized ASIC accelerator chiplets for DL applications. This approach offloads specific kernels of the computation (e.g., matrix-matrix or matrix-vector multiplications) to the more efficient specialized chiplets, while leveraging the FPGA fabric to interface to the outside world and to implement rapidly changing DL model components.

Configuration and Security

An FPGA's *configuration circuitry* loads the bitstream into the millions of SRAM cells that control the LUTs, routing switches and configuration bits in hard blocks. On power up, a configuration controller loads this bitstream serially from a source such as on-board flash. When a sufficient group of configuration bits are buffered, they are written in parallel to a group of configuration SRAM cells, in a manner similar to writing a (very wide) word to an SRAM array. This configuration circuitry can also be accessed by the FPGA fabric and embedded processor subsystems, allowing *partial reconfiguration* of one part of the device while another portion continues processing. For high-reliability applications, this configuration circuitry can also be used to continuously read back the programmed configuration of the device and compute a cyclic redundancy check (CRC) in order to detect if any configuration SRAM cells have been upset by soft errors (such as those induced by high energy radiation).

A complete FPGA application is very valuable intellectual property, and without *security measures* it could be cloned simply by copying the programming bitstream. To avoid this, FPGA CAD tools can optionally encrypt a bitstream, and FPGA devices can have a private decryption key programmed in by the manufacturer to be used by the configuration controller, making a bitstream usable only by a single customer who purchases FPGAs with the proper key.

Conclusion

FPGAs have evolved from simple arrays of programmable logic blocks and IOs interconnected via programmable routing into complex multi-die systems with many different embedded components such as BRAMs, DSPs, high-speed external interfaces, and system-level NoCs. The recent adoption of FPGAs in the HPC and datacenter domains, along with the emergence of new high-demand applications such as deep learning, is ushering in a new phase of FPGA architecture design. These new applications and the multi-user paradigm of the datacenter create opportunities for architectural innovation. At the same time, process technology scaling is changing in fundamental ways. Wire delay is scaling poorly which motivates rethinking programmable routing architecture. Interposers and 3D integration enable entirely new types of heterogeneous systems. Controlling power consumption is an overriding concern, and is likely to lead to FPGAs with more power-gating and more heterogeneous hard blocks. We do not claim to predict the future of FPGA architecture, except that it will be interesting and different from today!

References

Abdelfattah MS, Betz V (2013) The case for embedded networks on chip on field-programmable gate arrays. IEEE Micro 34(1):80–89

Abdelfattah MS et al (2015) Take the highway: design for embedded NoCs on FPGAs. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 98–107

Ahmed E, Rose J (2004) The effect of LUT and cluster size on deep-submicron FPGA performance and density. IEEE Trans Very Large Scale Integr (VLSI) Syst 12(3):288–298

Ahmed I et al (2019) FRoC 2.0: automatic BRAM and logic testing to enable dynamic voltage scaling for FPGA applications. ACM Trans Reconfig Technol Syst (TRETS) 12(4):1–28

Betz V, Rose J (1998) How much logic should go in an FPGA logic block? IEEE Des Test Comput 15(1):10–15

Betz V, Rose J (1999) FPGA routing architecture: segmentation and buffering to optimize speed and density. In: ACM International Symposium on FPGAs, pp 59–68

Betz V et al (1999) Architecture and CAD for deep-submicron FPGAs. Springer Science & Business Media. New York, USA

Bohr MT (1995) Interconnect scaling – the real limiter to high performance ULSI. In: Proceedings of International Electron Devices Meeting. IEEE, pp 241–244

Boutros A et al(2018) You cannot improve what you do not measure: FPGA vs. ASIC efficiency gaps for convolutional neural network inference. ACM Trans Reconfig Technol Syst (TRETS) 11(3):1–23

Boutros A et al (2018) Embracing diversity: enhanced DSP blocks for low-precision deep learning on FPGAs. In: IEEE International Conference on Field Programmable Logic and Applications (FPL), pp 35–357

Boutros A et al (2020) Beyond peak performance: comparing the real performance of AI-optimized FPGAs and GPUs. In: IEEE International Conference on Field-Programmable Technology (FPT), pp 10–19

Boutros A et al (2022) Architecture and application co-design for beyond-FPGA reconfigurable acceleration devices. IEEE Access 10:95067–95082

Caulfield AM et al (2016) A cloud-scale acceleration architecture. In: IEEE/ACM International Symposium on Microarchitecture (MICRO), pp 1–13

Chaware R et al (2012) Assembly and reliability challenges in 3D integration of 28 nm FPGA die on a large high density 65 nm passive interposer. In: IEEE Electronic Components and Technology Conference, pp 279–283

Cheah HY et al (2014) The iDEA DSP block-based soft processor for FPGAs. ACM Trans Reconfig Technol Syst (TRETS) 7(3):1–23

Chiasson C, Betz V (2013a) COFFE: fully-automated transistor sizing for FPGAs. In: IEEE International Conference on Field-Programmable Technology (FPT), pp 34–41

Chiasson C, Betz V (2013b) Should FPGAs abandon the pass gate? In: International Conference on Field-Programmable Logic and Applications, pp 1–8

Chromczak J et al (2020) Architectural enhancements in intel agilex FPGAs. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 140–149

Ebeling C et al (2016) Stratix 10 high performance routable clock networks In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 64–73

Eldafrawy M et al (2020) FPGA logic block architectures for efficient deep learning inference. ACM Trans Reconfig Technol Syst (TRETS) 13(3):1–34

Estrin G (1960) Organization of computer systems: the fixed plus variable structure computer. In: Western Joint IRE-AIEE-ACM Computer Conference, pp 33–40

Feng W et al (2018) Improving FPGA performance with a S44 LUT structure. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 61–66

Fowers J et al (2018) A configurable cloud-scale DNN processor for real-time AI. In: ACM/IEEE International Symposium on Computer Architecture (ISCA), pp 1–14

Gaide B et al (2019) Xilinx adaptive compute acceleration platform: versal architecture. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 84–93

Ganusov I, Devlin B (2016) Time-borrowing platform in the Xilinx ultrascale+ family of FPGAs and MPSoCs. In: IEEE International Conference on Field Programmable Logic and Applications (FPL), pp 1–9

Halfhill TR (2010) Tabula's time machine. Microprocess Rep 131:0–0

Hall M, Betz V (2020) From tensorflow graphs to luts and wires: automated sparse and physically aware CNN hardware generation. In: IEEE International Conference on Field-Programmable Technology (FPT), pp 56–65

Hutton M et al (2005) Efficient static timing analysis and applications using edge masks. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 174–183

Kapre N, Gray J (2017) Hoplite: a deflection-routed directional torus NoC for FPGAs. ACM Trans Reconfig Technol Syst (TRETS) 10(2):1–24

Karandikar S et al (2018) FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In: International Symposium on Computer Architecture (ISCA). . IEEE, pp 29–42

Krupnova H, Saucier G (2000) FPGA-based emulation: industrial and custom prototyping solutions. In: International Workshop on Field-Programmable Logic and Applications (FPL). . Springer, pp 68–77

Kuon I, Rose J (2007) Measuring the gap between FPGAs and ASICs. IEEE Trans Comput-Aided Des Integr Circuit Syst 26(2):203–215

LaForest CE et al (2012) Multi-ported memories for FPGAs via XOR. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 209–218

Lai B-CC, Lin J-L (2016) Efficient designs of multiported memory on FPGA. IEEE Trans Very Large Scale Integr (VLSI) Syst 25(1):139–150

Langhammer M, Pasca B (2015) Floating-point DSP block architecture for FPGAs. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 117–125

Langhammer M et al (2021) Stratix 10 NX architecture and applications. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 57–67
Lemieux G et al (2000) Generating highly-routable sparse crossbars for PLDs. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 155–164
Lemieux G et al (2004) Directional and single-driver wires in FPGA interconnect. In: IEEE International Conference on Field-Programmable Technology (FPT), pp 41–48
Lewis D et al (2003) The Stratix routing and logic architecture. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 12–20
Lewis D et al (2005) The Stratix II logic and routing architecture. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 14–20
Lewis D et al (2009) Architectural enhancements in Stratix-III and Stratix-IV. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 33–42
Lewis D et al (2013) Architectural enhancements in Stratix V. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 147–156
Lewis D et al (2016) The Stratix 10 highly pipelined FPGA architecture. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 159–168
Lockwood JW et al (2012) A low-latency library in FPGA hardware for high-frequency trading. In: Annual Symposium on High-Performance Interconnects (HOTI), pp 9–16
Meher PK et al (2008) FPGA realization of FIR filters by efficient and flexible systolization using distributed arithmetic. IEEE Trans Signal Process 56(7):3009–3017
Murray K et al (2013) Titan: enabling large and complex benchmarks in academic CAD. In: IEEE International Conference on Field-Programmable Logic and Applications (FPL), pp 1–8
Murray K et al (2020a) VTR 8: high-performance cad and customizable FPGA architecture modelling. ACM Trans Reconfig Technol Syst (TRETS) 13(2):1–55
Murray K et al (2020b) Optimizing FPGA logic block architectures for arithmetic. IEEE Trans Very Large Scale Integr (VLSI) Syst 28(6):1378–1391
Nasiri E et al (2015) Multiple dice working as one: CAD flows and routing architectures for silicon interposer FPGAs. IEEE Trans Very Large Scale Integr (VLSI) Syst 24(5):1821–1834
Nikolić S et al (2020) Straight to the point: intra- and intercluster LUT connections to mitigate the delay of programmable routing. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 150–160
Nurvitadhi E et al (2018) In-package domain-specific ASICs for intel Stratix 10 FPGAs: a case study of accelerating deep learning using TensorTile ASIC. In: IEEE International Conference on Field-Programmable Logic and Applications (FPL), pp 106–1064
Nurvitadhi E et al (2019) Why compete when you can work together: FPGA-ASIC integration for persistent RNNs. In: IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp 199–207
Papamichael MK, Hoe JC (2012) CONNECT: re-examining conventional wisdom for designing NoCs in the context of FPGAs. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 37–46
Parandeh-Afshar H et al (2012) Rethinking FPGAs: elude the flexibility excess of LUTs with and-inverter cones. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 119–128
Petelin O, Betz V (2016) The speed of diversity: exploring complex FPGA routing toplogies for the global metal layer. In: IEEE International Conference on Field-Programmable Logic and Applications (FPL), pp 1–10
Petersen MB et al (2021) NetCracker: a peek into the routing architecture of Xilinx 7-series FPGAs. In: International Symposium on Field-Programmable Gate Arrays (FPGA)
Putnam A et al (2014) A reconfigurable fabric for accelerating large-scale datacenter services. In: ACM/IEEE International Symposium on Computer Architecture (ISCA), pp 13–24
Qian T et al (2018) A 1.25 Gbps programmable FPGA I/O buffer with multi-standard support. In: IEEE International Conference on Integrated Circuits and Microsystems, pp 362–365
Rasoulinezhad S et al (2019) PIR-DSP: an FPGA DSP block architecture for multi-precision deep neural networks. In: IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp 35–44

Rasoulinezhad S et al (2020) LUXOR: an FPGA logic cell architecture for efficient compressor tree implementations. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 161–171

Rettkowski J et al (2017) HW/SW co-design of the HOG algorithm on a xilinx zynq SoC. J Parallel Distrib Comput 109:50–62

Ronak B, Fahmy SA (2015a) Mapping for maximum performance on FPGA DSP blocks. IEEE Trans Comput-Aided Design Integr Circuits Syst 35(4):573–585

Ronak B, Fahmy SA (2015b) Minimizing DSP block usage through multi-pumping. In: International Conference on Field Programmable Technology (FPT)

Sivaswamy S et al (2005) HARP: hard-wired routing pattern FPGAs. In: International Symposium on Field-Programmable Gate Arrays (FPGA)

Swarbrick I et al Network-on-chip programmable platform in versal ACAP architecture. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 212–221

Tang X et al (2019) A study on switch block patterns for tileable FPGA routing architectures. In: IEEE International Conference on Field-Programmable Technology (FPT), pp 247–250

Tatsumura K et al (2016) High density, low energy, magnetic tunnel junction based block RAMs for memory-rich FPGAs. In: IEEE International Conference on Field-Programmable Technology (FPT), pp 4–11

Tessier R et al (2007) Power-efficient RAM mapping algorithms for FPGA embedded memory blocks. IEEE Trans Comput-Aided Des Integr Circuits Syst 26(2):278–290

Turakhia Y et al (2018) Darwin: a genomics co-processor provides up to 15,000x acceleration on long read assembly. ACM SIGPLAN Not 53(2):199–213

Tyhach J et al (2004) A 90 nm FPGA I/O buffer design with 1.6 Gbps data rate for source-synchronous system and 300 MHz clock rate for external memory interface. In: IEEE Custom Integrated Circuits Conference, pp 431–434

Upadhyaya P et al (2016) A fully-adaptive wideband 0.5–32.75 Gb/s FPGA transceiver in 16 nm FinFET CMOS technology. In: IEEE Symposium on VLSI Circuits, pp 1–2

Wang E et al (2019) Deep neural network approximation for custom hardware: where we've been, where we're going. ACM Comput Surv (CSUR) 52(2):1–39

Wilton S et al (1995) Architecture of centralized field-configurable memory. In: ACM International Symposium on Field-Programmable Gate Arrays (FPGA), pp 97–103

Wong H et al (2011) Comparing FPGA vs. custom cmos and the impact on processor microarchitecture. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 5–14

Yazdanshenas S, Betz V (2018) Interconnect solutions for virtualized field-programmable gate arrays. IEEE Access 6:10497–10507

Yazdanshenas S, Betz v (2019) COFFE 2: automatic modelling and optimization of complex and heterogeneous FPGA Architectures. ACM Trans Reconfig Technol Syst (TRETS), 12(1):1–27

Yazdanshenas S et al (2017) Don't forget the memory: automatic block RAM modelling, optimization, and architecture exploration. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 115–124

Yiannacouras P et al (2009) Data parallel FPGA workloads: software versus hardware. In: IEEE International Conference on Field-Programmable Logic and Applications (FPL), pp 51–58

Young-Schultz T et al (2020) Using openCL to enable software-like development of an FPGA-accelerated biophotonic cancer treatment simulator. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 86–96

Zgheib G et al (2014) Revisiting and-inverter cones. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 45–54

Zhao Z et al (2020) Achieving 100 Gbps intrusion prevention on a single server. In: USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp 1083–1100

Coarse-Grained Reconfigurable Array (CGRA)

14

Zhaoying Li, Dhananjaya Wijerathne, and Tulika Mitra

Contents

Introduction 466
Historical Context 468
Architecture: A Landscape of Modern CGRA 471
Compilation for CGRAs 475
Modulo Scheduling and Modulo Routing Resource Graph (MRRG) 475
CGRA Mapping Approaches 477
Other Compilation-Related Issues 492
Conclusions 500
References 500

Abstract

Coarse-grained reconfigurable array (CGRA) is a promising class of spatial accelerator that offers high performance, energy efficiency, as well as flexibility to support a wide range of application domains. CGRAs can bridge the gap between efficient but inflexible domain-specific accelerators and flexible but inefficient general-purpose processors. A CGRA is essentially an array of word-level processing elements connected via on-chip interconnect. Both the processing elements and the interconnect can be reconfigured per cycle following the on-chip configuration memory content. Thus the compiler needs to map the compute-intensive loop kernels of the application onto the CGRA in a spatio-temporal fashion by setting up the configuration memory. The simplicity and parallelism of the architecture coupled with the efficacy of the compiler enable the CGRA to reach the dual goal of hardware-like efficiency with software-like programmability. We present a comprehensive review of the CGRAs starting

Z. Li · D. Wijerathne · T. Mitra (✉)
National University of Singapore, Singapore, Singapore
e-mail: zhaoying@comp.nus.edu.sg; dmd@comp.nus.edu.sg; tulika@comp.nus.edu.sg

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_50

with the historical context, sketching the architectural landscape, and providing an extensive overview of the compilation approaches.

Keywords

CGRA · Reconfigurable architecture · Spatial accelerator

Introduction

The history of computing has been dominated by general-purpose processors (Hennessy and Patterson 2011) that can execute any possible application offering unlimited flexibility. Unfortunately, such processors suffer from low performance and energy efficiency due to the high overhead involved in executing the instructions beyond just the computation (e.g., fetching instructions and data from the memory, decoding instructions, respecting the data and control dependencies, etc.) (Hameed et al. 2010) and extracting parallelism from sequential instructions stream at runtime. At the other end of the spectrum, we are witnessing the emergence of domain-specific hardware accelerators (Dally et al. 2020) for many popular tasks such as deep neural networks, image/video processing, cryptography, among others. These ASIC (application-specific integrated circuit) accelerators provide high performance and energy efficiency but zero flexibility as they are tied to one specific task or application domain. Reconfigurable spatial accelerators present a compromise between the two extremes by supporting ASIC-like efficiency while maintaining flexibility through software programmability (Mitra 2015).

A coarse-grained reconfigurable array (CGRA) (Wijerathne et al. 2022a) is a spatial hardware accelerator with very simple architecture. A generic CGRA comprises a 2D array of processing elements capable of performing basic arithmetic, logic, and memory operations at word level using the functional unit (FU) and a small register file (RF) as temporary data storage as shown in Fig. 1. Each processing element is connected to its neighbors through the switch and can transfer the result of the computation to selected neighbors for the next cycle. Both the

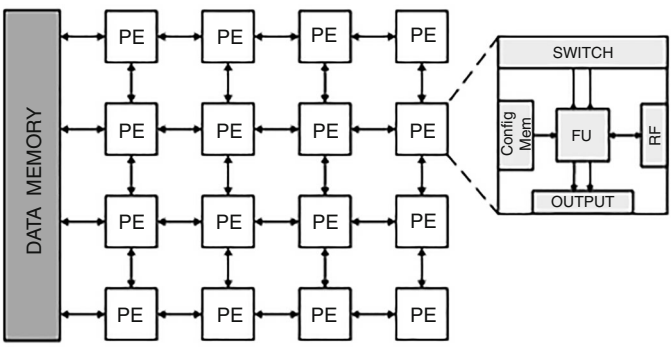


Fig. 1 A classic 4 × 4 CGRA (coarse-grained reconfigurable array)

computation performed by each individual processing element and the routing of the data to the neighbors through the interconnect can be configured on a per-cycle basis. This is achieved by storing a predetermined sequence of configurations for a limited number of cycles in an on-chip memory (configuration memory). At runtime, the sequence of configurations is repeated in a cyclical fashion. In other words, the CGRA fabric can be configured in both the spatial (restricted by the number of processing elements) and temporal (restricted by the number of configurations that can be stored on chip) domains. In addition, a new sequence of configurations can be brought into the CGRA from external storage, if necessary, at the cost of runtime delay. We will explore the variations of CGRA architectures in section "Architecture: A Landscape of Modern CGRA."

The high performance of the CGRA comes from the parallelism offered by the large number of on-chip processing elements. On the other hand, the simplicity of the architecture that just faithfully follows the planned computation and routing (generated by the compiler) without any runtime effort to extract parallelism from the application leads to significantly improved energy efficiency compared to the general-purpose processors. At the same time, the word-level (coarse-grained) reconfigurations supported by the CGRA compared to bit-level (fine-grained) reconfigurations of the FPGAs (Field-Programmable Gate Arrays) empower the CGRAs to achieve higher performance and lower power compared to the FPGAs. Finally, the per-cycle temporal configuration dimension of the CGRA is a powerful feature that allows the CGRA to operate with smaller spatial dimensions by time-multiplexing computation and dataflow as opposed to only spatial dimension in the FPGA. The reconfiguration in FPGA, while possible, can only happen over a longer time interval similar to bringing in a new sequence of configurations in CGRA. Thus, CGRA accelerators can have smaller chip area and hence lower power (specially leakage power) compared to the FPGAs. Obviously, the complexity burden is now transferred from the architecture to the compiler.

Let us now focus on how the compiler can exploit the spatio-temporal configuration of the CGRA to accelerate application execution. As the CGRA execution repeatedly cycles through a limited-length sequence of configurations, the application loop kernels are the perfect candidates for acceleration. The compiler is responsible for extracting as much parallelism from the loop kernel as possible (subject to data dependency constraints) and maximizing utilization of the array of processing elements. This will lead to reduced temporal length of the configuration sequence and significantly improved runtime of the kernel.

The CGRA compiler achieves the mapping by embracing the dataflow computing model. In this model, the compiler exposes all the computations and the flow of data between dependent computations from the high-level sequential code fragment of the loop kernel. Figure 2 shows a dataflow graph of the general matrix multiply (GEMM) kernel (Kågström et al. 1998). This dataflow graph is subsequently mapped onto the CGRA to maximize parallelism while satisfying all the constraints of the architecture as well as data dependencies within the loop kernel. The challenge now is to map the computations within the loop kernel onto the processing elements by finding appropriate spatio-temporal coordinates and route the data dependencies between processing elements. Figure 3 shows the spatio-temporal

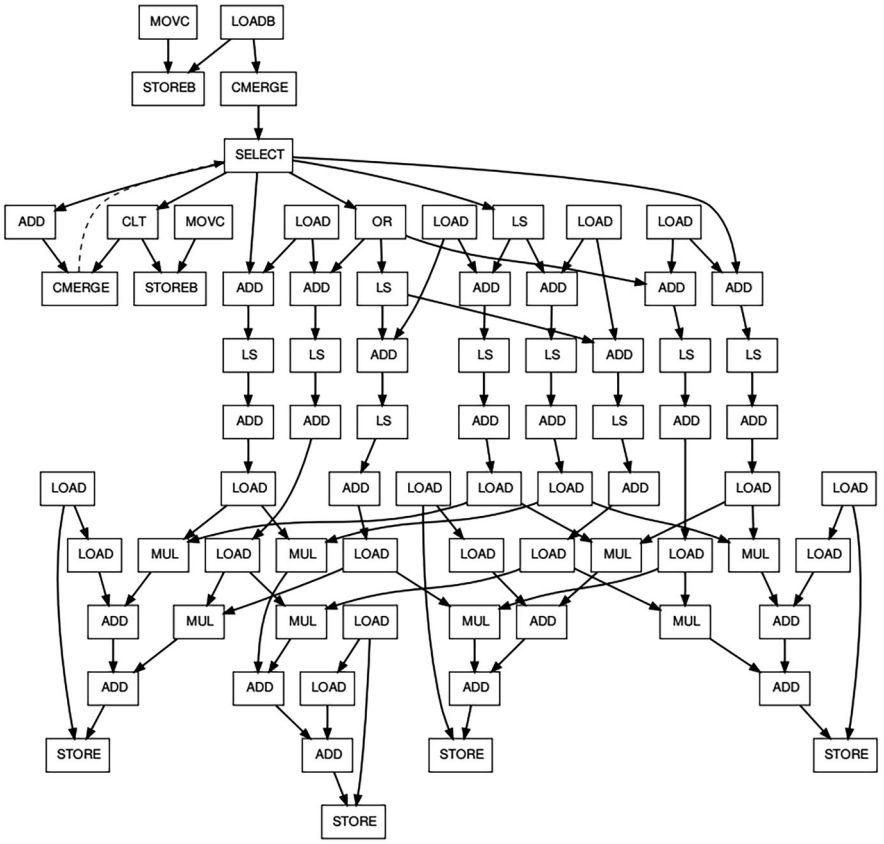


Fig. 2 Dataflow graph of the general matrix multiply (GEMM) kernel

mapping of a simple dataflow graph on a 2 × 2 CGRA. We provide an in-depth tour of the diverse mapping approaches with trade-offs between the quality of the mapping and the compilation time in section "Compilation for CGRAs."

Finally, we introduce additional challenges and opportunities in the CGRA accelerator space in terms of data memory management, configuration memory management, and mapping of entire application as opposed to a single isolated loop kernel.

Historical Context

The first general-purpose microprocessor, Intel 4004, was introduced in 1971. The microprocessor industry since then has enjoyed an unprecedented growth in performance due to Moore's Law (Moore et al. 1998), Dennard scaling (Dennard et al. 1974), and micro-architectural innovations (Hennessy and Patterson 2011).

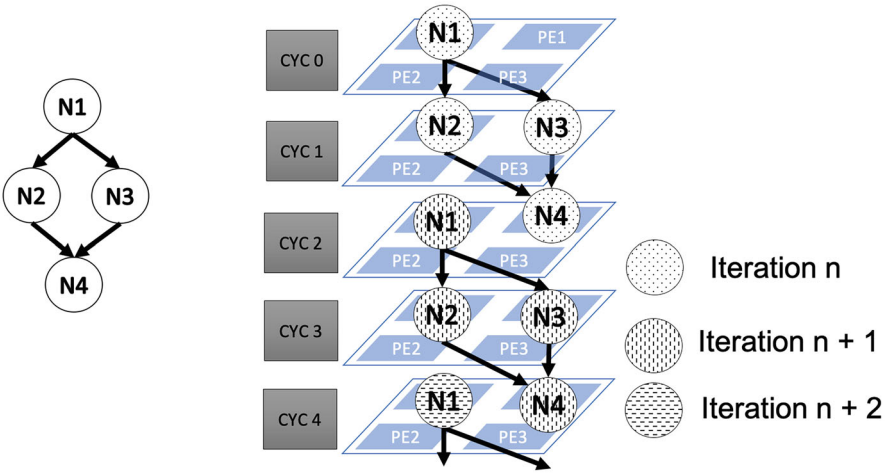


Fig. 3 Spatio-temporal mapping of a simple dataflow graph on a 2 × 2 CGRA

While Moore's law was responsible for the sustained increase in clock frequency, the processor performance improved further due to several micro-architectural innovations including processor pipeline, out-of-order execution, speculation, and cache memory hierarchy among others. These advancements enabled the processor to extract instruction-level parallelism (ILP), thereby boosting the critical instructions-per-cycle (IPC) metric (Hennessy and Patterson 2011). More importantly, as the ILP was extracted transparently by the underlying architecture from single-threaded programs, the software developers enjoyed the performance benefit without any additional effort. Together, the growth in clock frequency and IPC ensued the relentless gain in processor performance spanning over three decades. However, this performance growth has come to an end with power wall due to the breakdown of Dennard scaling, ILP wall, and memory wall (Patterson et al. 2006). Thus, computing systems made the irreversible transition in early 2000 toward multi- and many-core architectures to gainfully employ the growing number of transistors supported by Moore's law and exploit thread-level parallelism (TLP) instead of ILP. However, simply increasing the core count in multi-cores is no longer tenable as the sequential fragment limits the speedup of the entire application according to Amdahl's law (Amdahl 1967).

Against this backdrop, domain-specific accelerators (Dally et al. 2020; Jouppi et al. 2017; Ghorpade et al. 2012; Rashid et al. 2019) specialized for a particular task such as deep neural networks, image/video processing, encryption, etc., have become prevalent from tiny Internet-of-Things (IoT) devices to the datacenters. Current system-on-chips (SoCs) include a number of special-purpose accelerators. Shao et al. (2015) analyzed die photos from three generations of Apple's SoCs: A6 (iPhone 5), A7 (iPhone 5S), and A8 (iPhone 6) to show that consistently more than half of the die area is dedicated to application-specific hardware accelerators and

estimated the presence of around 29 accelerators in A8 SoC. The ITRS roadmap predicts hundreds to thousands of customized accelerators by 2022 (Carballo 2014). These tailor-made ASIC (application-specific integrated circuit) accelerators can provide excellent performance and energy efficiency but suffer from lack of flexibility as they are restricted to only one particular task. Thus, such accelerators can only be feasible for tasks that are ubiquitous across multiple applications.

Ideally, we want the best of both the worlds, i.e., a general-purpose universal accelerator that can reach close to the performance and efficiency of domain-specific accelerators while maintaining software programmability and flexibility to support multiple tasks. Reconfigurable computing (Compton and Hauck 2002) fills this gap between hardware and software with far superior performance potential compared to programmable cores while maintaining higher level of flexibility than ASICs.

Field-programmable gate arrays (FPGAs) are pre-fabricated semiconductor devices that can be reprogrammed to create almost any digital circuit/system (Kuon et al. 2008). FPGAs contain an array of computation elements, called configurable logic blocks connected through a set of programmable routing resources. A digital circuit can be fabricated on FPGAs by appropriately setting the configuration bits of the logic blocks for the functionality and connecting these blocks together through reconfigurable routing. This comes at the cost of area, power, and delay: an FPGA requires approximately 20 to 35 times more area than ASIC, has roughly 3–4 times slower performance than ASIC, and consumes about 10 times as much dynamic power (Kuon and Rose 2007).

Coarse-grained reconfigurable arrays (CGRAs) (Choi 2011) are promising alternative between ASICs and FPGAs. FPGAs do not have as high efficiency as ASIC accelerators due to the fine bit-level granularity of reconfiguration that results in lower performance, higher energy consumption, and longer reconfiguration penalty. In contrast, CGRAs, as the name suggests, comprise coarse-grained functional units (FUs) connected via typically a mesh-like interconnect as shown in Fig. 1. The functional units are capable of performing arithmetic/logic operations and can be reconfigured on a per-cycle basis by writing to a control (context) register associated with each functional unit. The functional units can exchange data among themselves through the interconnect. As many functional units work in parallel, CGRAs can easily accelerate compute-intensive loop executions by exploiting instruction-level parallelism. The primary challenge lies with the compiler that needs to map and schedule the instructions on the FUs as well as take care of the routing of data among the FUs through the interconnect.

CGRA was introduced around 2000 (Singh et al. 2000; Mei et al. 2003a; Baumgarte et al. 2003). Recently, CGRA is witnessing a resurgence in both industry and academia as a promising accelerator architecture that can provide both efficiency and programmability. DARPA has recently launched software-defined hardware (SDH) program (Darpa software defined hardware 2019) to build CGRA-like reconfigurable hardware that would enable near ASIC performance without sacrificing programmability. Various CGRAs are appearing from academia and industry, such as HRL (Gao and Kozyrakis 2016), Plasticine (Prabhakar et al. 2017), HyCUBE (Karunaratne et al. 2017), Wave DPU (Nicol 2017), Sambanova (Emani

et al. 2021), Samsung Reconfigurable Processor (Suh et al. 2012), Renesas Dynamically Reconfigurable Processor (DRP) (Fujii et al. 2018), and Intel Configurable Spatial Accelerator (Fleming et al. 2020). These CGRAs have more processing elements and complex architectures compared to the original designs and thus require more compilation effort to efficiently utilize the hardware resources.

There have been many works on domain-specific spatial accelerators in recent literature (Chen et al. 2019; Jouppi et al. 2017; Lu et al. 2017; Tu et al. 2017; Kwong and Chandrakasan 2011; Yoo et al. 2012). These accelerators target applications in specific domains such as deep neural network, image analysis, and signal processing. The micro-architecture of domain-specific spatial accelerators shares many similarities with CGRAs. Like CGRAs, most of the domain-specific accelerators have an array of processing elements connected in a two-dimensional grid. However, the processing elements have limited and specific computation capability. The interconnection network is designed to support specific dataflow and not fully reconfigurable. For example, in the Google Tensor Processing Unit (TPU), the processing elements only support multiply and accumulation operations, while the interconnection network supports systolic dataflow for matrix multiplication (Jouppi et al. 2017). These domain-specific accelerators can be viewed as different instantiation of domain-agnostic CGRA accelerator that can be configured in software to support any dataflow and computation.

Architecture: A Landscape of Modern CGRA

In this section, we provide a brief overview of the basic CGRA architecture and its variations. For a detailed survey of the CGRA architectures, the readers can refer to Liu et al. (2019) and Podobas et al. (2020).

Basic CGRA architecture A CGRA consists of a set of processing elements (PE) and an on-chip network. A CGRA can reconfigure each PE for different operations and the network for different routing on a per-cycle basis. Figure 1 shows an abstract block diagram of a classic 4×4 CGRA. It uses a 2D mesh network, and each PE is connected to its neighboring PEs. A PE comprises a functional unit (FU), register file (RF), crossbar switches, and configuration memory. Each FU can have one or more ALU (arithmetic-logic unit) or other computation units. The on-chip data memory, usually scratchpad memory (SPM), feeds data to the whole PE array. The data transfer between the SPM and the off-chip memory takes place through direct-memory access (DMA). In each cycle, a PE reads a configuration from the configuration memory and configures the corresponding modules such as the ALU, the switches, and the RF ports. Then PE executes the operation and passes the data to other PEs through the on-chip network.

Homogeneous and Heterogeneous CGRA From the perspective of the PEs, the CGRAs can be classified into two categories: homogeneous and heterogeneous. In homogeneous CGRA, all the PEs have the same functionality, while in

heterogeneous CGRA, the PEs can have different functionalities. If a CGRA targets application kernels from some specific domains, special PEs can be useful, such as the ones supporting multiply-accumulate (MAC) operations in machine learning. However, if these special PEs are costly in terms of area or power, then the CGRA includes special functionality in only some of the PEs. Most CGRAs provide heterogeneity in terms of memory access functionality. For example, in the CGRA of Fig. 1, it is not necessary to let all the PEs access the on-chip data memory. The latency for data memory access is generally much longer than computation, and the SPM also has a limited number of ports restricting the number of parallel accesses. Hence, usually only the PEs at the boundary can access the SPM. Another example is RAPID architecture (Venkataramani et al. 2019) that has a 1D array of special function units (SFUs) alongside a 2D array of PEs. The SFU is used to support FP32 operations, and other PEs can only support integer operations.

A recent work REVAMP (Bandara et al. 2022) proposes a generalized automated approach in heterogeneous CGRA exploration that can work across diverse architectures. It is a design space exploration framework that can automatically realize more power-efficient heterogeneous CGRA versions from a given homogeneous CGRA and target application suit. Their micro-architectural optimizations cover a broad scope of heterogeneity, including compute, interconnect, and PE-local storage. It also automatically generates compiler support to map loop kernels onto derived heterogeneous architecture efficiently.

Spatial CGRA A CGRA can reconfigure the PEs for different operations and routing per cycle. Each PE is associated with a configuration memory. The configuration memory stores a limited number of configuration words, one per cycle. The PE rotates or loops through these configuration words and accordingly sets the operation of the FU and the routing for the switches and the RF. A special case is a CGRA with only one configuration word and is referred to as a spatial CGRA. A spatial CGRA can reduce area, power, as well as cycle time (higher clock frequency) as there is no reconfiguration delay involved. The area and power of the configuration memory are considerable for the CGRAs. In Karunaratne et al. (2018), the power consumption of a 4 KB configuration memory in a 4×4 CGRA is around 40% of the whole chip power. A spatial CGRA is more energy-efficient than traditional CGRAs. However, it does not have the advantage of temporal dimension and essentially reduces to FPGA but with coarse-grained reconfigurable units. Note that the limited configuration memory, while area- and energy-efficient, may not be able to accommodate large kernels or need loop partitioning with runtime configuration reloading to accelerate such kernels.

On-chip network The on-chip network connects the PEs to route data. In each PE, there are routing paths from the input ports to the output ports. Also, the data can be stored in the register file while waiting for processing or further routing. The most common network is neighbor-to-neighbor (N2N) connection. Each PE is connected to its neighboring PEs, and neighbors can be reached in one cycle.

Routing to distant PEs requires other intermediate PEs and needs multiple cycles. The simple N2N network, however, provides very limited interconnection on the chip. It needs tremendous compilation effort to achieve good speedup in accelerating kernels with complex data dependencies and even then the speedup can be limited.

A recent CGRA architecture called HyCUBE (Karunaratne et al. 2017) creates a larger virtual neighborhood for each PE by allowing single-cycle multiple-hop connections. HyCUBE designs a special bypass network to allow intermediate PEs to forward data to other PEs without consuming the data. Thus a PE can send data to distant PEs in one cycle. The HyCUBE chip (Wang et al. 2019) offers four hops per cycle at a maximum clock frequency of 753 MHz. Increasing the number of hops per cycle further will reduce the maximum possible clock frequency. The dataflow graph (DFG) of an application kernel can have complex structure and data dependencies. While N2N networks need multiple cycles when the source and destination nodes corresponding to a data dependency are mapped to distant PEs, HyCUBE only needs one cycle to route most data dependencies leading to better performance in terms of both compilation time (as the mapping becomes easier with larger neighborhood) and actual kernel execution time (due to reduced delay in routing data dependencies). Moreover, in N2N network, a PE that is involved in routing transient data cannot perform computation in the same cycle as the data need to be stored in the register file of the PE requiring an explicit move operation. HyCUBE allows the intermediate PEs in the bypass path to continue executing operations leading to better utilization of the PEs in performing useful computation.

The above networks cannot scale well with increasing CGRA sizes. A bigger CGRA is usually tiled into blocks, and each block is a small sub-CGRA. The network among the blocks often has a higher bandwidth than the one inside a block. An example of such titled architecture is Plasticine (Prabhakar et al. 2017) that provides scalar and vector communication channels between the blocks.

Memory hierarchy Typically, the CGRA memory hierarchy consists of two types of memory: data memory to hold input, output, and intermediate data and the configuration memory to hold the configuration directives for the FU, RF, and the switches.

Most CGRA architectures use multi-bank scratchpad memory as the global on-chip data memory (Mei et al. 2003a; Singh et al. 2000; Karunaratne et al. 2017). Scratchpad memories are fully software-controlled, meaning that the data movement between the off-chip main memory and on-chip scratchpad memory is explicitly controlled through directives generated by the compiler. Therefore, scratchpad memories are more power-efficient than hardware-controlled caches. Multi-bank memory SPM is used to increase the data throughput, i.e., the number of parallel accesses between the SPM data memory and the PE array. Usually, each memory bank has a few (one or two) read/write ports, and a subset of PEs have access to each memory bank. The CGRA PEs execute load and store operations to load the input data and store the computed data back into the on-chip memory.

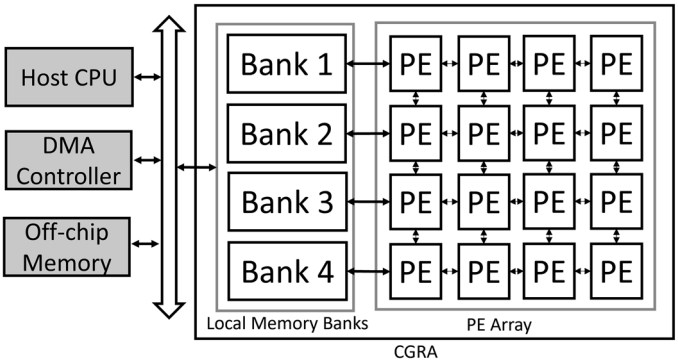


Fig. 4 On-chip memory hierarchy of CGRA loosely coupled with host CPU

Figure 4 shows CGRA data memory with four memory banks where only the boundary PEs on the left side have access to the data memory. Some architectures perform the load/store address generation within the PE array, while others have specialized hardware address generation units (Wijerathne et al. 2019). Apart from global data memory, some CGRA architectures use shared register files to hold intermediate data. These register files are shared between a subset of PEs. It provides an alternative to the on-chip network for communication between those subsets of PEs.

The CGRA configuration memory, also referred to as context/instruction memory, holds the directives for CGRA execution each cycle including the operation to be executed by the PEs and the routing configurations for the crossbars switches. As CGRAs are specifically used for accelerating loop kernels, the same sequence of configurations are repeated over a fixed number of cycles. The configurations are loaded into the configuration memory before the CGRA execution starts. The configuration memory can be either centralized (global) or decentralized (local), where each PE has a separate configuration memory. Even in a decentralized setting, the configurations for the PEs are fetched and decoded in a lockstep manner. Therefore, program counters of all the PEs have the same value even though they have different configurations.

Interface Between CPU and CGRA

CGRAs are used to accelerate compute-intensive loop kernels of the applications. Therefore, it needs to be coupled with the host processor for executing a complete application. The host processor is responsible for running the non-loop code, configuring the CGRA, and initiating the DMA data transfers from the main memory to the CGRA local memory.

Some CGRAs are closely coupled with the main processor, where CGRA is a part of the main CPU. For example, ADRES (Mei et al. 2003a) CGRA is tightly coupled with the main processor, where the top row of the PE array is a VLIW processor

that acts as the main processor. Figure 4 shows a loosely coupled CPU where the CGRA is connected to an independent accelerator. MorphoSys CGRA (Singh et al. 2000) is an example of a loosely coupled CGRA. Loosely coupled CGRAs offer more flexibility in the design phase as they can be designed independently. In a loosely coupled system, both the CPU and the CGRA can execute code in parallel in a non-blocking manner. A tightly coupled system typically cannot execute code in parallel on the CPU and the CGRA as they share the same resources. However, the overheads in data transfer are higher in the loosely coupled system compared to the tightly coupled system.

Compilation for CGRAs

Given a loop from an application and a CGRA architecture, the goal of compilation is to map the loop onto the CGRA (i.e., generate the configurations for a fixed number of cycles) to maximize the throughput. In general, this compilation is referred as *mapping* in the CGRA world. The loop is represented as a dataflow graph (DFG), where the nodes represent the operations and the edges represent the dependency between the nodes.

Modulo Scheduling and Modulo Routing Resource Graph (MRRG)

Modulo Scheduling Modulo scheduling is a software pipelining technique to exploit the instruction-level parallelism among the loop iterations (Rau 1994). There are often inadequate instruction-level parallelism in a single iteration of a loop. Pipelining consecutive loop iterations can provide more parallelism and thus improve the resource utilization. Figure 5a shows a 2×2 homogeneous CGRA, and we assume each PE can support any operation. Figure 5b shows an example DFG where each node represents an operation, such as addition, multiplication, etc. Mapping of the DFG onto the CGRA has two components: placement and routing. The placement decides which PE will execute each operation, and routing makes sure that the data can be routed to the dependent operations in a timely manner.

Figure 5c shows a possible mapping of the DFG in Fig. 5b onto the CGRA in Fig. 5a. For the sake of convenience, the 2×2 CGRA in Fig. 5a has been drawn as a linear array. The mapping has three parts: prologue, steady-state kernel, and epilogue. The prologue and epilogue are executed only once at the start and end of the loop execution. The steady-state kernel is repeated and includes all the operations from one or more iterations. The schedule length of the kernel is called the initial interval (II) and indicates the number of cycles between the initiation of consecutive loop iterations. For a loop with a large number of iterations, the execution time is dominated by the II value.

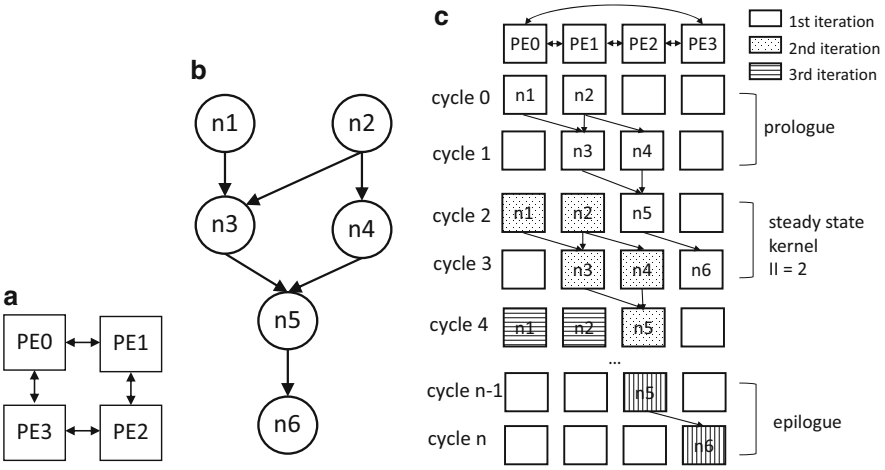


Fig. 5 2×2 CGRA, a DFG (dataflow graph), and the mapping. (**a**) 2×2 CGRA. (**b**) DFG example (**c**) DFG mapping example

In the mapping of Fig. 5c, II = 2. Notice that node *n5* of the first loop iteration is executing in the same cycle with *n1* and *n2* from the second-loop iteration. Hence, the CGRA can start a new loop iteration every two cycles leading to II value of two. The routing is done through the network among the PEs. This figure shows an abstract mapping for convenience. A real mapping will include the detailed routing configuration at each PE.

Given a DFG and a CGRA, the mapper first calculates the minimum initial interval (MII), which is the maximum of the resource-minimal II and the recurrence-minimal II. The resource MII depends on the number of PEs and the number of DFG nodes (assume one PE can process one DFG node). Hence, the resource MII cannot be less than the number of DFG nodes divided by the number of PEs. The recurrence MII is determined by the dependency across loop iterations. Let us assume that we have an operation $a[i] = a[i-1] \times b[i]$. The operation of iteration i must wait for the result of the operation of last iteration $i-1$. The recurrence MII can be calculated by traversing the DFG.

Mapping a compute-intensive loop kernel of an application to CGRAs using modulo scheduling was first discussed in the DRESC compiler (Mei et al. 2003b). The algorithm starts with an II equal to the maximum between the resource-minimal II and recurrence-minimal II and attempts to schedule the loop. If it fails, it tries with successively larger II values.

Modulo Routing Resource Graph (MRRG) Mei et al. (2003b) proposed the MRRG, which represents the resources and the routing for a time-extended CGRA. The nodes in MRRG represent the ports of the register file, the on-chip network, the ALU inside PE, etc. The edges are the connections among the CGRA components represented as nodes. The MRRG is a directed graph G_{II} where II corresponds to

the initiation interval. Given a graph G, let us denote the vertex set by $V(G)$ and the edge set by $E(G)$. Each node $v \in V(G_{II})$ is a tuple *(n, t)*, where n refers to the resource in CGRA and t is the cycle ($0 \leq t \leq II - 1$). Let $e = (u, v) \in E(G_{II})$ be an edge where $u = (m, t)$ and $v = (n, t + 1)$. Then the edge e represents a connection from resource m in cycle t to resource n in cycle $t+1$. In general, if resource m is connected to resource n in the CGRA, then node $u = (m, t)$ is connected to node $v = (n, t + 1), t \geq 0$.

Figure 6 shows the MRRG corresponding to the CGRA in Fig. 5a when the II is 2. The resources of 2×2 CGRA are replicated every cycle along the time axis. In modulo scheduling, if a node $v = (n, t)$ in the MRRG becomes occupied, then all the nodes $v' = (n, t + k \times II)$ (where $k > 0$) are also occupied. For example, in Fig. 5c, *PE0* is occupied by node *n1* at cycle 0 and the II is 2. Thus the node will occupy *PE0* every $2 \times k$ cycle. Hence, after cycle 1, the configuration in cycle 0 will be used to reconfigure the fabric, as the II is 2 and configuration has two items. Thus, there are wrapped around edges from the second item back to the first one, as cycle 3 will use the first configuration item. These edges show hardware resource connection along the time axis.

CGRA Mapping Approaches

In this section, we present three broad classes of mapping approached based on heuristics, mathematical optimization, and graph-theory-inspired techniques.

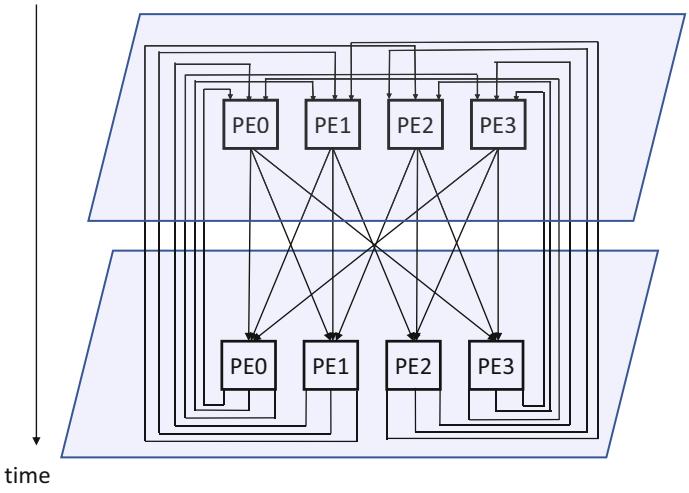


Fig. 6 An example of modulo routing resource graph (MRRG)

Heuristic Approaches

The heuristic approaches propose customized solutions for the CGRA mapping problem.

Simulated Annealing

Meta-heuristics are problem-independent approaches that treat the architectural elements as black boxes. Simulated annealing is one of the most popular meta-heuristic methods. Here, we introduce the usage of simulated annealing in CGRA mapping as proposed in the DRESC compiler (Mei et al. 2003b). For a target II value, the algorithm first generates an initial schedule satisfying the dependence constraints but with possibly over-subscribed resources. For example, more than one operation might be scheduled on the same functional unit in the same cycle. The algorithm then iteratively reduces resource overuse and tries to come up with a legal scheduling via simulated annealing that explores different placement and routing options until a valid placement and routing of all operations and data dependencies are found. The cost function used during the simulated annealing is based on the total routing cost, i.e., the combined resource consumption of all the placed operations and the routed data dependencies. In this technique, a huge number of possible routes are evaluated. As a result, the technique has long convergence time, especially for large dataflow graphs.

Routing through the register files and the register allocation problems are further explored in De Sutter et al. (2008), which extends the work in Mei et al. (2003b). Register allocation is achieved by constraining the register usage during the simulated annealing place and route process. The imposed constraint is adopted from the meeting graph (Eisenbeis et al. 1995) for solving loop cyclic register allocation in VLIW processors. In post-routing phase, the registers are allocated by finding a Hamilton circuit in the meeting graph, which is solved as a traveling salesman problem (De Sutter et al. 2008). This technique is specially designed for CGRAs with rotating register files. Hatanaka and Bagherzadeh (2007) and CGRA-ME (Chin et al. 2017) follow the simulated annealing framework but aim at finding better cost functions for over-used resources.

Edge-Centric Modulo Scheduling

The DRESC compiler performs node-centric modulo scheduling where the nodes are scheduled and placed first followed by routing of the edges. In contrast, for edge-centric modulo scheduling (EMS) (Park et al. 2008a), the

(continued)

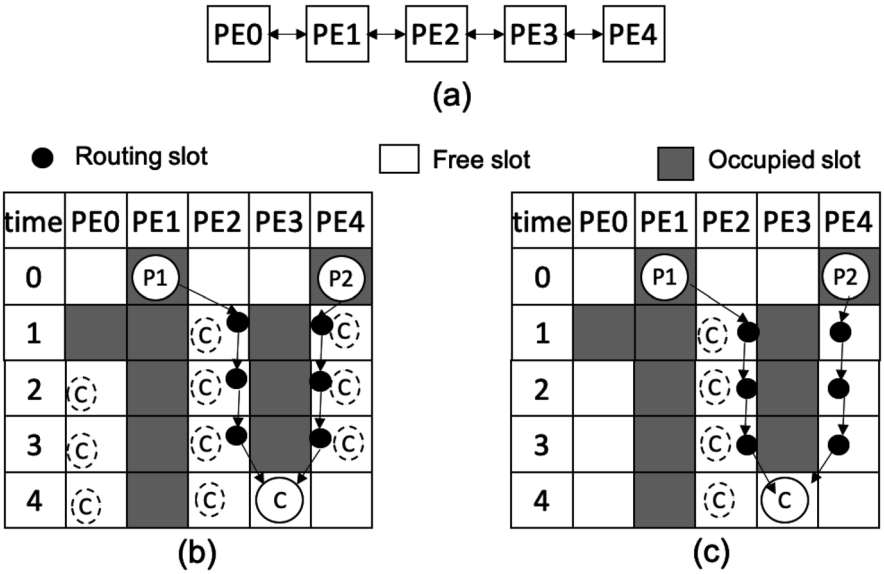


Fig. 7 Node-centric (left) versus edge-centric (right) modulo scheduling (Park et al. 2008a)

primary objective is routing efficiency rather than operations assignment. Figure 7 taken from Park et al. (2008a) shows the difference between the two approaches.

Node-centric approaches place an operation according to the heuristic routing cost. The cost consists of various metrics that reflects the quality of the mapping. The mapper visits the PE candidates and selects the best one or visits the candidates one by one until it finds a solution. When visiting a candidate, the mapper will try to route the edges from the mapped nodes to the current candidate. Figure 7b shows how an optimal placement is found with this approach. A DFG including two producers P1 and P2 and a shared consumer C is mapped onto a 1×5 CGRA in Fig. 7a. P1 and P2 are already placed, and the mapper places the consumer C by visiting all the empty slots as shown in Fig. 7b. The slots with dashed circles are failed attempts as the mapper cannot establish routing from producers P1 and P2. After visiting those slots, the mapper successfully places C on PE3 at time 4 and routes values from P1 and P2.

In an edge-centric approach, the routing function contains the placement of an operation, and the placement decision is made when the routing information is discovered. When scheduling an operation, the mapper picks an edge from the operation's previously placed producers or consumers and starts

(continued)

routing the edge. The router will search for an empty slot that can execute the target operation. Once a suitable slot is found, the mapper will place the operation and route for other edges.

Figure 7c shows the same example of Fig. 7b, and the consumer is mapped using an edge-centric approach. The scheduler tries to route edge from P1 to C first, instead of placing operation C directly. When an empty slot is found, the scheduler temporarily places the target operation and checks if there are other edges connected to the consumer; if so, it recursively routes those edges. For example, when the router visits slot (PE2,1), it temporarily places C there and recursively calls the router function to route the edge from P2 to C. When it fails to route the edge from P2 to C, routing resumes from slot (PE2,1), and not from P1, and a solution is eventually found at slot (PE3,4).

In general, an edge-centric approach can find a solution faster and achieves better quality mapping compared to a node-centric approach. However, it has a greedy nature in that it optimizes for a single edge at a time, and the solution can easily fall into local minima. There is no search mechanism in the scheduler at the operation level, and every decision made in each step is final. This problem can be addressed by employing intelligent routing cost metrics as priorities. The quality of a mapping using specific priorities highly depends on efficient heuristics for assigning these priority values to both the operations and the resources.

Schedule, Place, and Route (SPR)

SPR (Friedman et al. 2009) is a mature CGRA mapping tool that successfully combines the VLIW-style scheduler and FPGA placement and routing algorithms for CGRA application mapping. It consists of three individual steps namely scheduling (ordering operations in time based on data and control dependencies), placement (assigning operations to functional units), and routing (mapping data signals between operations using wires and registers). SPR uses iterative modulo scheduling (IMS) (Rau 1994), simulated annealing (Kirkpatrick et al. 1983) placement with a cooling schedule inspired by VPR (Betz and Rose 1997), and PathFinder (McMurchie and Ebeling 2008) and QuickRoute (Li and Ebeling 2008) for pipelined routing.

IMS is a VLIW-inspired loop instruction scheduling algorithm. IMS heuristically assigns operations to a schedule specifying the start time for each instruction, taking into account resource constraints and data and control dependencies. SPR uses IMS for initial operation scheduling and extends IMS to support rescheduling with feedback from the placement algorithm, letting it to handle the configurable interconnects of CGRAs.

(continued)

FPGA mapping tools historically use simulated annealing for placement and PathFinder for routing. VPR (Betz and Rose 1997), which has become the de facto standard for FPGA architecture exploration and is similar to SPR in that it seeks to be a flexible and open mapping tool that can provide high-quality mappings and support a wide spectrum of architectural features. Unfortunately, it only applies to FPGAs. SPR adopts similar algorithms but extended for CGRAs to support multiplexing of resources across cycles and solving the placement and routing issues that arise when using a fixed frequency device. SPR uses QuickRoute to solve the pipelined routing problem.

List Scheduling
A list scheduling algorithm is adapted in the mapping algorithms of Bansal et al. (2003). Priority-based list scheduling heuristic is used in Bansal et al. (2003) to map data-dependent operations in the kernel onto spatially close PEs in the CGRA. Each operation in the kernel is mapped onto a PE considering the operation priority and ability to route data from already mapped parent operations. They maintain a PE list based on topology traversal order and an operation list based on scheduling priority. Topology traversal order is the order in which PEs are traversed in the CGRA while mapping operations to PEs. The experiment results show that spiral traversal order performs best. According to a scheduling priority, the operation list is maintained, which gives preference to operations on the longest data dependency paths. Operation with the highest priority is mapped on the next available PE in the PE list if there is a valid route from already mapped parent operations. Scheduling is done on a cycle-by-cycle basis. Each cycle, the algorithm schedules operations on each PE and then increments the cycle when the PE list is exhausted. This process is continued until all the operations in the kernel have been scheduled. Unfortunately, the list scheduling algorithms do not produce a software pipelined schedule and are thus unable to exploit inter-iteration parallelism.

Evolutionary Algorithm
The mapping approach in Lee et al. (2011) presents a fast heuristic using a quantum-inspired evolutionary algorithm. This evolutionary algorithm uses an initial solution obtained from list scheduling as a starting point. The algorithm uses Q-Bit encoding to represent the hundreds of possible mapping results and evaluates each case to choose the best solution that satisfies the

(continued)

data dependency constraints. Q-Bit encoding allows compact maintenance of potential mappings, enabling fast design space exploration compared to other evolutionary algorithms. Fitness function is the performance, which is the inverse of the total latency. The algorithm iteratively improves the solution until it finds a solution with the lower bound of optimal latency or there is no improvement during a given time interval. However, the experimental evaluation is limited to small loop kernels with few DFG operations and CGRAs with limited reconfigurability.

Machine Learning

A reinforcement-learning-based mapping approach for CGRAs has been proposed in RLMap (Liu et al. 2018). The CGRA mapping problem is formulated as an agent in reinforcement learning. Each mapping state is represented as a distinct image that captures operation placement and inter-PE routing. Agent action is defined as the interchange of operations on neighbor PEs to keep the action space small. The reward function is defined based on a cost function that captures interconnect power requirements, utilized compute PEs, routing PEs, and empty PEs. Reward function helps the agent obtain valid and high-quality mapping in terms of power, area, and performance.

Inspired by the progress in deep learning, Yin et al. (2017a) proposed DFGNet, a convolutional neural-network-based mapping approach. They present dual-input neural network to capture kernel DFG and CGRA architecture. CGRA mapping problem is translated into an image-based classification problem in a convolutional neural network. Input DFG is represented as an adjacency matrix, and a matrix represents the CGRA architecture state. The neural network consists of convolutional, max pooling, concatenate, and fully connected layers. The issue with any deep learning method for application mapping on CGRAs is the difficulty in obtaining the abundant training data required for such approaches.

CGRAs differ in the network and the PE function. Existing compilers (Wang et al. 2019; Hamzeh et al. 2013; Dave et al. 2018) usually leverage special characteristics of the architecture to generate quality mapping. These compilers, however, are usually hand-crafted, making it challenging from the time-to-market perspective. Li et al. (2022) proposed a portable framework, LISA, to map DFGs onto diverse CGRAs. LISA uses graph neural network (GNN) to analyze dataflow graph (DFG) to derive *labels* that describe how the DFG should be mapped, e.g., the estimated routing resource required by an edge and the predicted mapping distance between the DFG nodes. With trained GNNs, these labels can reflect characteristics of the accelerator.

(continued)

Moreover, LISA provides a global view in mapping by describing the whole DFG and accelerator characteristics. For a new accelerator, the portable compiler re-trains the GNN model to adapt the labels according to the accelerator characteristics.

Mathematical Optimization Techniques

We now present two mathematical optimization techniques for CGRA mapping.

Integer Linear Programming (ILP)

ILP-based formalization of the CGRA mapping problem has been proposed in the literature (Ahn et al. 2006; Chin and Anderson 2018). The ILP formulation consists of all the requirements and the constraints that must be satisfied by a valid schedule. The formulation is built from the DFG and the MRRG and hence highly portable as shown recently in the CGRA-ME project (Chin and Anderson 2018). However, it is not clear whether the ILP modeling can be effective for all possible architectural features, and more importantly, scalability is a huge issue with ILP techniques that can only be applied to very simple loop kernels.

Boolean Satisfiability (SAT) Solvers

A SAT-solver-based application mapping approach for CGRAs has been proposed by Wave Computing for their CGRA architecture (Chaudhuri and Hetzel 2017). This technique has been demonstrated to automatically compile dataflow programs onto a commercial massively parallel CGRA-based dataflow processor (DPU) containing 16,000 processing elements. This is an innovative application of Boolean satisfiability to formally solve this complex and irregular optimization problem and produce high-quality results comparable to hand-written assembly code produced by human experts. The approach is reported to be efficient in utilizing processing elements with rich micro-architectural features such as complex instructions, multi-precision data paths, local memories, register files, switches, etc. However, the approach requires custom algorithms to handle the complexity of the SAT-based solutions in offering scalable, robust technique. A constraint-based approach is also used for the Silicon Hive CGRA architecture (Burns et al. 2004) though the details are not publicly available.

Graph-Theory-Inspired Techniques

Many CGRA mapping approaches use graph theory concepts to formulate the CGRA mapping problem. Those approaches transform the CGRA mapping problem into well-known graph-theoretic formulations and leverage the existing techniques to solve the problem. This section categorizes the graph-theory-inspired mapping techniques based on the graph theory formalism they use. We also discuss, in more detail, prominent CGRA mapping techniques that correspond to each formalization. Table 1 summarizes different aspects of five such notable works.

Following graph theory, concepts are widely used to formalize and solve CGRA application mapping problem:

1. Subgraph homeomorphism
2. Graph epimorphism
3. Graph minor
4. Compatibility graph
5. Graph drawing

To understand the above graph theory concepts, we need to first present few related definitions. Therefore, let us first look at the definitions of graph isomorphism, graph subdivision, graph homeomorphism, and induced subgraph.

Definition 1. A directed graph $G = (V, E)$ is a pair where V is a set of vertices and $E \subseteq V \times V$ is a set of edges. Let G and G' be two graphs where $G = (V, E)$ and $G' = (V', E')$.

Definition 2. Graph isomorphism: An isomorphism from G to G' is a bijective function $f : V \to V'$ such that $(u, v) \in E \iff (f(u), f(v)) \in E'$.

Table 1 Notable works using graph theory concepts for CGRA mapping problem

Work	Graph theory concept	Solution	What is new?
Alle et al. (2008)	Homeomorphism	Greedy algorithm for transformation	Mapping DFG substructures
EpiMap (Hamzeh et al. 2012)	Epimorphism	Heuristic-based search	Recomputation to solve out-degree problem
Graph Minor (Chen and Mitra 2014)	Graph Minor	Tree search method	Allow route sharing
RegiMap (Hamzeh et al. 2013)	Compatibility graph	Finding a max clique	Allow both route sharing, recomputation
SPKM (Yoon et al. 2009)	Graph Drawing	Split and push approach	Support heterogeneous architectures

Two graphs are isomorphic when both graphs contain the same number of vertices connected in the same way. Figure 8 shows two isomorphic graphs. Graph isomorphism is an equivalence relation on directed graphs.

Definition 3. Graph Subdivision: The subdivision of some edge $e = (u, v) \in E$ yields a graph containing one new vertex w and with an edge set replacing e by two new edges, (u, w) and (w, v).

The definition of graph subdivision is self-explanatory. In Fig. 9, graph H is formed by subdivision of graph G.

Definition 4. Graph Homeomorphism: Two graphs G and G' are homeomorphic if there is a graph isomorphism from some subdivision of G to some subdivision of G'. In general, a subdivision of a graph G is a graph resulting from the subdivision of edges in G.

In Fig. 10, graph H can be created by subdivision of edges of G and also by subdivision of edges of G'. Therefore, graph G and graph G' are homeomorphic.

Definition 5. Induced Subgraph: Let $U \subseteq V$ be a subset of vertices of G. The subgraph of G induced by U is $G_{\downarrow U} = (U, E \cap (U \times U))$.

Induced subgraph is a graph formed by a subset of vertices of another graph with all of the edges that connect the vertices in that subset.

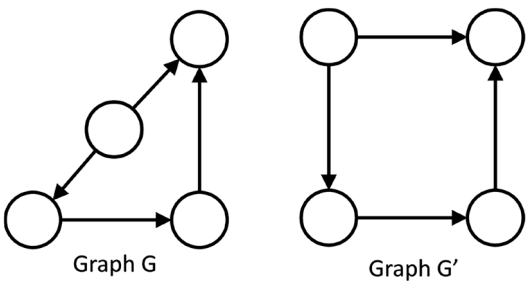


Fig. 8 Graph isomorphism

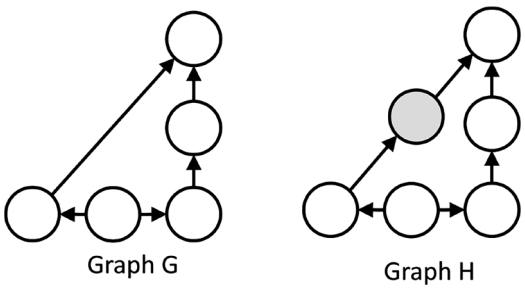


Fig. 9 Graph subdivision

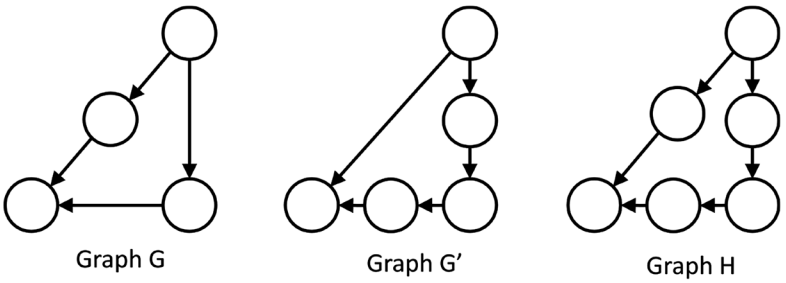


Fig. 10 Graph homeomorphism

Subgraph-Homeomorphism-Based Techniques
Formal definition of subgraph homeomorphism is as follows:

Definition 6. Subgraph Homeomorphism: A subgraph homeomorphism from G to G' is a homeomorphism f from an induced subgraph $G_{\downarrow U}$ of G to G'.

Let G be a directed graph representing the DFG and H_{II} be a directed graph representing the MRRG with initiation interval II. In the ideal scenario of full connectivity among the PEs, we can map all the data dependencies in the DFG to direct edges in the MRRG. That is, for any edge $e = (u, v) \in E(G)$, there is an edge $e = (f(u), f(v)) \in E(H)$, where f represents the vertex mapping function from the DFG to the MRRG. This matches the definition of subgraph isomorphism. However, data may need to be routed through a series of nodes rather than direct links in reality. If an edge $e = (u, v) \in E(G)$ in the DFG can be mapped to a path from $f(u)$ to $f(v)$ in the MRRG H, it matches the subgraph homeomorphism definition. The idea is to test if the DFG G representing the loop kernel is subgraph homeomorphic to the MRRG H_{II} representing the CGRA resources and their interconnects.

Figure 11 illustrates the subgraph homeomorphism formulation. Figure 11a shows a simple DFG being mapped onto a 2×2 homogeneous mesh CGRA shown in Fig. 11b. The DFG is homeomorphic to the subgraph of the MRRG shown in Fig. 11c, and thus the subgraph represents a valid mapping (for simplicity, we have removed additional nodes of the MRRG). In this homeomorphic mapping, edges (1, 3) and (1, 4) have been routed through three additional routing nodes marked by R. Notice that each routing node has degree 2 and has been added through edge subdivision (marked by dashed edges). Alternatively, we can smooth out the routing nodes in the MRRG subgraph to obtain the original DFG.

(continued)

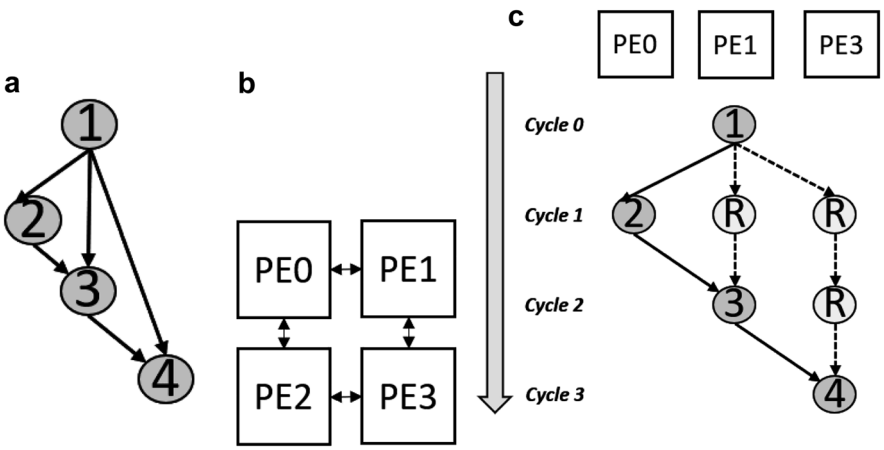


Fig. 11 Subgraph homeomorphism formulation of CGRA mapping problem. (**a**) DFG (**b**) CGRA (**c**) MRRG

The subgraph homeomorphism techniques for CGRA mapping problem has been adopted in Tuhin and Norvell (2008), Alle et al. (2008), and Brenner et al. (2009). Authors in Tuhin and Norvell (2008) formulate the mapping problem as finding a node disjoint subgraph homeomorphism between the DFG and the MRRG. The mapping algorithm is adapted from Modulo Scheduling with Integrated Register Spilling (MIRS) (Zalamea et al. 2001), a modulo scheduler capable of instruction scheduling with register constraints. Alle et al. (2008) partition the DFG into subgraphs called HyperOps, and these HyperOps are synthesized into hardware configurations. The synthesis is carried out through a homeomorphic transformation of the dependency graph of each HyperOp onto the hardware resource graph. They employ a greedy algorithm for the transformation. Brenner et al. (2009) also formalize the CGRA mapping as a subgraph homeomorphism problem. However, they consider general application kernels rather than loops.

However, subgraph homeomorphism requires the edge mappings to be node disjoint (except at end points) or edge disjoint (Fortune et al. 1980). As a result, subgraph-homeomorphism-based techniques exclude the possibility of sharing the routing nodes among single-source multiple target edges (Park et al. 2008b) (also called multi-net), leading to possible wastage of precious routing resources.

Graph-Epimorphism-Based Technique
Graph epimorphism is defined based on graph homomorphism. Therefore, let us first look at the definition of graph homomorphism. A graph homomorphism defines a mapping between two graphs in which adjacent vertices in the first graph are mapped to adjacent vertices in the second graph. Unlike isomorphism, homomorphism can be from a bigger to a smaller graph. The formal definition of a homomorphism is as follows:

Definition 7. Graph Homomorphism: A homomorphism from G to G' is a function $f : V \to V'$ such that $(u, v) \in E \implies (f(u), f(v)) \in E'$.

Graph epimorphism relaxes the bijection constraint of graph isomorphisms to a surjection constraint on both vertices and edges (hence the terminology of epimorphism). Several vertices of G may be mapped on the same vertex of G'.

Definition 8. Graph Epimorphism: An epimorphism from G to G' is a surjective function $f : V \to V'$ such that:

- If $(u, v) \in E \implies (f(u), f(v)) \in E'$ (graph homomorphism).
- If $(u', v') \in E'$, then there exists $(u, v) \in E$ such that $f(u) = u'$ and $f(v) = v'$ (surjectivity on edges).

EPIMap (Hamzeh et al. 2012) formalizes the CGRA mapping problem as a graph epimorphism problem with the additional feature of recomputation. Recomputation allows for the same operation to be performed on multiple PEs if it leads to better routing. In the EPIMap approach, the DFG G is morphed into another graph G' (through the introduction of routing/recomputation nodes and other transformations) such that there exists subgraph epimorphism from G' to G (many to one mapping of vertices from G' to G and adjacent vertices in G' map to adjacent vertices in G). Then EPImap attempts to find the maximal common subgraph (MCS) between G' and the MRRG graph H using standard MCS identification procedure. If the resulting MCS is isomorphic to G', then a valid mapping has been obtained; otherwise, G is morphed differently in the next iteration and the process repeats. EPIMap can generate better scheduling results compared to EMS with a similar compilation time. Figure 12 taken from Hamzeh et al. (2012) shows the benefit of recomputation. The mapping in Fig. 12d computes the node b in both PE1 and PE2 at cycle 1. This recomputation results in a better mapping (II=2) compared to mapping without recomputation (II=3) in Fig. 12c.

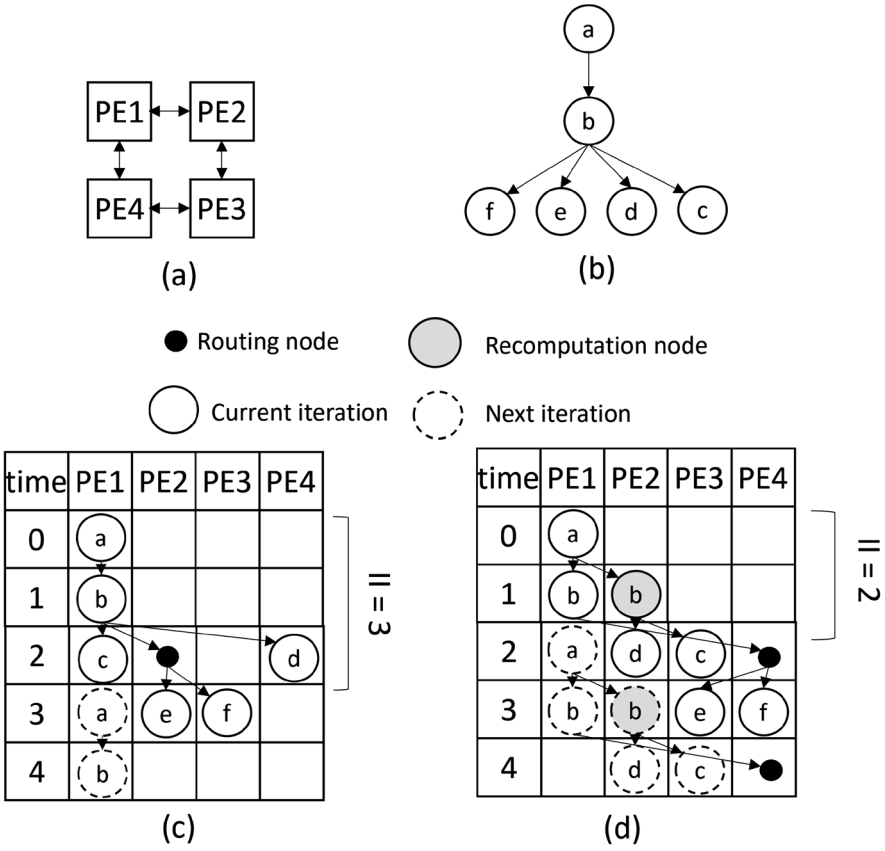


Fig. 12 A valid mapping without using recomputation (left) versus with recomputation (right) in EpiMap (Hamzeh et al. 2012)

Graph-Minor-Based Technique

Definition 9. Graph Minor: An undirected graph G is called a minor of the graph G' if G is isomorphic to a graph that can be obtained by zero or more edge contractions on a subgraph of G'. An edge contraction is an operation that removes an edge from a graph while simultaneously merging together the two vertices it used to connect. A model of G in G' is a mapping ϕ that assigns to every edge $e \in E$ an edge $\phi(e) \in E'$, and to every vertex $v \in V$ a non-empty connected tree subgraph $\phi(v) \subseteq G'$ such that:

(continued)

- The graphs $\phi(v)|v \in V$ are mutually vertex disjoint, and the edges $\phi(e)|e \in E$ are pairwise distinct.
- If $e = (u, v) \in E$, the edge $\phi(e)$ connects $\phi(u)$ with $\phi(v)$.

G is isomorphic to a minor of G' if and only if there exists a model of G in G'.

Graph minor (Chen and Mitra 2014) models the CGRA mapping problem as a graph minor containment problem that can explicitly model route sharing. As explained in the definition, a graph H is a minor of graph G if H can be obtained from a subgraph of G by a (possibly empty) sequence of edge contractions (Robertson and Seymour 1990). The graph minor is initially defined for undirected graphs, but the authors in Chen and Mitra (2014) adapt the definition to directed graphs for CGRA mapping. In this context, we need to test if the DFG is a minor of the MRRG, where the edges to be contracted represent the routing paths in the MRRG. The mapping algorithm is inspired by the tree search method, which is widely used to solve graph matching problems. Unlike edge subdivision (or its reverse operation smoothing), edge contractions are not restricted to simple paths. Thus graph minor formalism naturally allows for route sharing. Figure 13 shows the difference between mappings under graph minor approach (Fig. 13d) and subgraph homeomorphism approach (Fig. 13c). The number of routing nodes is reduced from 3 (in subgraph homeomorphism mapping) to 2 (in graph minor mapping) through route sharing.

Compatibility-Graph-Based Technique

REGIMap (Hamzeh et al. 2013) presents a general formulation of the problem of mapping a kernel on the CGRA while using its registers to minimize II. The formulation partitions the problem into a scheduling problem and an integrated placement and register allocation problem. They first create a compatibility graph, a subgraph of the product of the DFG G and MRRG H. The vertices of the compatibility graph denote the operation–resource pair, which represent possible mapping pairs. The edges of the graph denote the compatibility of two corresponding mapping pairs. The mapping problem is reduced to one of finding the largest clique in the compatibility graph under the constraints of register resources. Then an efficient and constructive heuristic is used to solve the mapping problem.

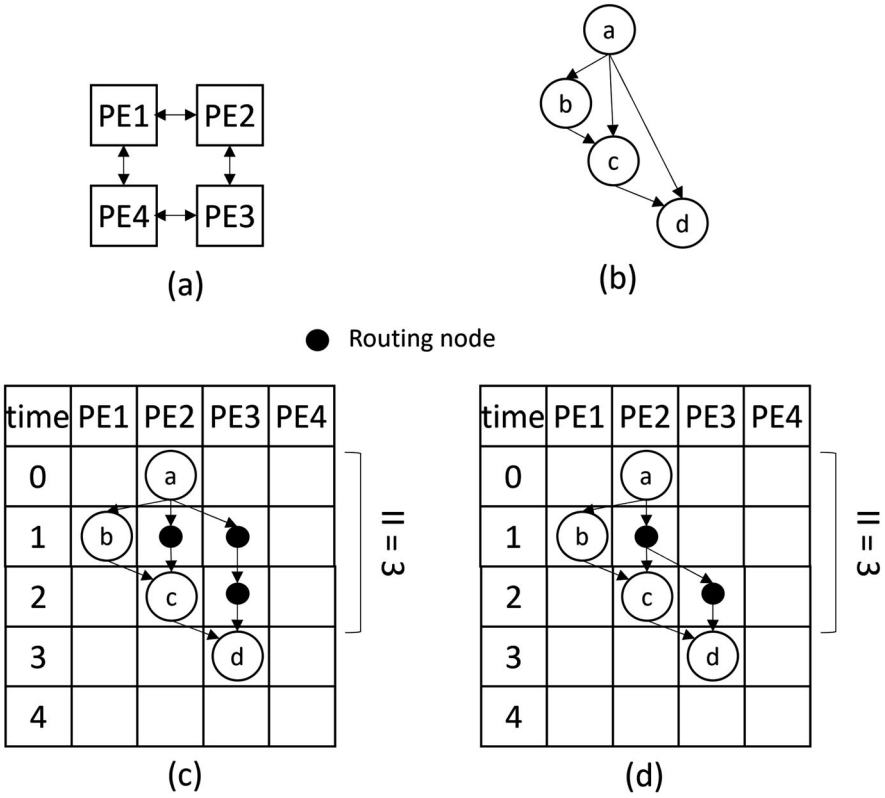


Fig. 13 Subgraph homeomorphism (left) versus graph minor formulation (Chen and Mitra 2014) (right) of CGRA mapping problem

Graph-Drawing-Based Technique
SPKM (Yoon et al. 2009) adopts the split and push technique (Di Battista et al. 1998) for planar graph drawing and focuses on spatial mappings for CGRAs. The mapping in SPKM starts from an initial drawing where all DFG nodes reside in the same group. One group represents a single processing element. The group is then split into two, and a set of nodes are pushed to the newly generated group. The split process continues until each group contains only one node, representing a one-to-one map ping from DFG to the planar resource graph of CGRA.

Other Compilation-Related Issues

Challenges Related to Data Access

The works presented for computation mapping have largely ignored the impact of data placement in the on-chip memory and the communication between the CPU and CGRA. Those works mostly assume data are already present in the local data memory. They also assume all PEs have access to all data memories, i.e., infinite memory bandwidth between local data memory and PE array. However, in reality, CGRA local memory bank has non-uniform memory access architecture where only a subset of the PEs have access to a memory bank with a limited number of read/write ports (Kim et al. 2010). Even when CGRA mapping achieves higher compute utilization under the assumption of ideal memory, the memory limitations could cause overall performance degradation in the actual setting. Thus, the compiler should be aware of the data memory limitations to minimize the effects of the memory bottleneck.

Figure 14a shows the simplified DFG of array-addition loop kernel. The kernel adds elements in two arrays $A[]$, $B[]$ and stores the results in array $C[]$. Shaded nodes represent memory access operations: two load operations (L) and one store operation (S). Memory address of each array ($\&A[i]$, $\&B[i]$, $\&C[i]$) is computed in the nodes above the L/S nodes based on the iteration variable i. Figure 14b shows the mapping of the DFG on CGRA coupled with on-chip local memory with four banks. Only boundary PEs have access to a directly connected memory bank. Arrays $A[]$, $B[]$, and $C[]$ are placed in memory banks 1, 2, and 4, respectively. The CGRA mapper should be aware of the data placement to correctly place the load/store operations on the PEs. Therefore, data placement and CGRA mapping

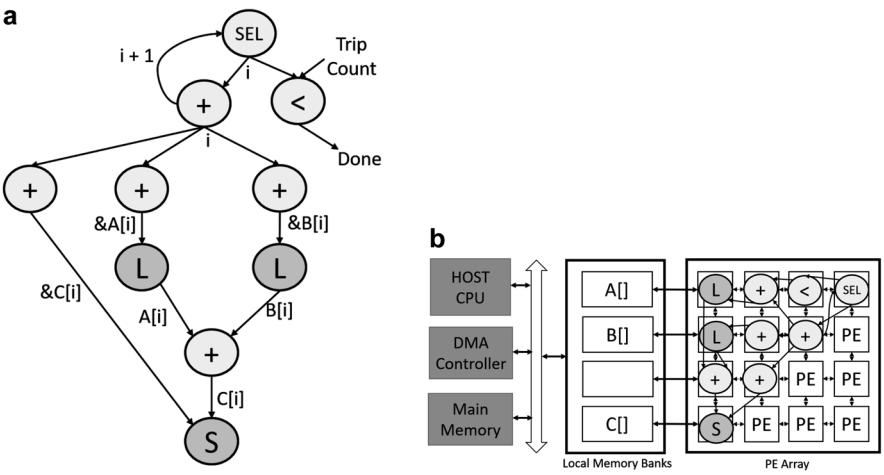


Fig. 14 Memory-aware loop mapping on CGRA. (**a**) DFG of the array-addition loop kernel. (**b**) Mapping of array-addition loop kernel

are interdependent tasks. Host CPU manages the data movement using a DMA controller based on the data placement decided by the compiler.

This section discusses compiler-based solutions for challenges related to data access in CGRA.

Memory-Aware Compilation
Effective memory-aware compiling strategy should:

- Place the data without under-utilizing the memory banks.
- Consider the limited connections between the PE array and the memory banks.
- Prevent memory access conflicts.
- Maximize data reuse by avoiding data duplication.

Kim et al. (2010) propose a memory-aware mapping solution that considers the effects of various memory architecture parameters, including the number of banks, local memory size, and the communication bandwidth between the local memory and the external main memory. Their heuristic-based mapping approach considers minimizing duplicate arrays, balancing bank utilization, and balancing computation and data transfer time.

Memory access conflict arises when the number of data accesses per bank per cycle is higher than the number of memory ports in one memory bank. Such access conflicts can be resolved either by data duplication or hardware queue with arbiters. Both approaches result in higher cost in terms of performance and power. A better solution would be to let the compiler partition the data into memory banks to avoid access conflicts. The application mapping technique in Kim et al. (2010) considers the memory banking architecture and maps operations and data to avoid memory bank conflicts. The initial schedule is generated without considering the data mapping. Subsequently, it uses array clustering and conflict-free scheduling until a conflict-free mapping is found. They also consider a hardware solution called DAMQ (Dynamically Allocated, Multi-queue buffer), which uses a request queue and arbiter to resolve access conflicts. This hardware approach increases the access latency of the memory operation. They show the software solution is 8.5% better than a hardware solution. Yin et al. (2016a, 2017b) also propose memory access conflict-free loop mapping strategy and a joint mapping flow by integrating modulo scheduling and memory partitioning. Dual-force directed scheduling algorithm is designed to solve the CGRA mapping problem and memory partitioning problem jointly.

When supporting kernels with multiple accesses for the same data array, a naive data placement in multi-bank memory could result in many access conflicts. Wang et al. (2013) propose a memory partitioning scheme for multi-

(continued)

dimensional arrays based on a linear transformation. It partitions the multi-dimensional array among different banks to place each parallelly accessed data element in a separate memory bank.

Yin et al. (2015) propose memory-aware mapping technique that uses shared data memory as the routing resource. They argue that routing some data dependencies through memory could improve the performance. Therefore, data dependencies that consume multiple PE routing resources are replaced by memory access operations. They divide the mapping problem into two subproblems: (1) replacing the dependence with memory access operations and (2) integrating placement and routing with the PE allocation for memory operations. Then, those two subproblems are solved to find a valid mapping. They establish a precise formulation for the CGRA mapping problem while using shared local data memory as a routing resource and present a practical approach for mapping loops to CGRAs.

Memory Address Generation

One other main challenge related to data access is the way data memory addresses are generated. Wijerathne et al. (2019) show a substantial amount of instructions in loop kernels correspond to address generation (ranging from 20% to 80%). One solution is to offload the address generation to specialized address generation units since address generation involves a common operation pattern.

Nowatzki et al. (2017) and Wijerathne et al. (2019) advocate the separation of execution and memory address generation due to the overhead of address generation in CGRA. Nowatzki et al. (2017) propose a decoupled access–execute CGRA with complex on-chip memory hierarchy and a stream-based programming interface. In decoupled access–execute CGRA model, memory address generation is decoupled from the execution of main computation. Wijerathne et al. (2019) propose a decoupled access–execute CGRA with software and hardware support for conflict-free memory access.

Nested Loop Mapping

The application mapping approaches presented in the previous subsections consider a single innermost loop. In the recent literature, several works explore the mapping of loop nests beyond the innermost level.

There are two main motivations for going beyond the innermost loop level. First, nested loops offer more parallelism than what is available in a single innermost loop level. Therefore, going beyond the innermost loop level could improve the available instruction-level parallelism. Second, the host processor needs to invoke the CGRA

multiple times to support imperfect nested loops when only the innermost loop body is mapped to the CGRA. Multiple invocations lead to overheads, including pipeline filling/draining and the initialization of loop variables and pipeline parameters on the CGRA.

Mapping Approaches
We categorize existing works for nested loop mapping based on their approach.

Polyhedral-Model-Based The polyhedral model is a robust framework that is widely used as a loop transformation technique. Polyhedral-based transformations can be applied to nested loops where the loop bounds and array references are affine functions of loop iterators and program parameters. Liu et al. (2013) use the polyhedral model to map the innermost two loops of multi-level loop nests. They use the polyhedral model to transform the two-dimensional nested loops to a new iteration domain with a parallelogram shape. Then they tile the parallelogram into multiple tiles where each tile consists of numerous iterations in the original program. Operators in each tile are mapped to CGRA using a place and route algorithm. The objective of the problem formulation is to reduce the total execution time by determining tile parameters. They adopt genetic algorithm to solve the problem.

Loop Flattening Based Loop flattening can convert imperfect loop nests into a single nested loop and can be executed in a single invocation. However, loop flattening comes with the overhead of increased code size. Lee et al. (2014) argue that overhead from increased code size is lower than the overheads from multiple invocations. To limit the negative impact of loop flattening, they combine loop fission with flattening and introduce few specialized operations to CGRA PEs called nested iterators, extended accumulator, and periodic store.

Systolic-Mapping-Based HiMap (Wijerathne et al. 2021a,b) proposes a hierarchical mapping approach to map regular multi-dimensional kernels on CGRAs. They use systolic mapping (Lee and Kedem 1990) as an intermediate abstraction layer to guide the hierarchical mapping. Each iteration in the multi-dimensional iteration space is mapped to a virtual PE cluster on CGRA based on a space-time mapping matrix derived from systolic mapping algorithm. Then operations in each iteration are mapped to physical CGRA PEs. They only generate detailed mapping for the few unique iterations with unique computation and routing patterns. The mappings of the unique iterations are replicated to obtain the final valid mapping. Therefore, HiMap is fast and scalable for mapping regular multi-dimensional kernels.

Nested Loop Mapping Under Limited Configuration Memory
Yin et al. (2016b) propose a method to map two of the innermost levels of loop nests using less configuration memory compared to Lee et al. (2014). In this context, configuration memory size constraint presents a significant limitation. To solve this configuration memory capacity constraint, recent works Cao et al. (2017) and Jafri et al. (2015) propose architectural improvements to use the configuration memory as a cache that stores the most recently accessed loop nests at runtime. The dynamic caching leads to performance improvement because more application segments can be accelerated, and the data transfer between the host and the CGRA is minimized. It is possible to naively employ caching within a loop nest to expand the mappable loop nests beyond the innermost loops. Still, the frequent context switching between outer and inner loops may incur significant overhead. DNestMap (Karunaratne et al. 2018), a partitioning and mapping tool for CGRAs, can judiciously extract the most beneficial code segments of multiple deeply nested loops and effectively cache them together statically in the configuration memory through spatio-temporal partitioning.

Application-Level Mapping

An application usually has several kernels, which can be a sequential code, a single loop, or multiple loops, or any combination of them. CGRA can reconfigure the functionality of PE and the routing of on-chip network to accelerate any kernel. Application-specific integrated circuits (ASICs) always target-specific kernels and lose the flexibility to process other kernels. Field-programmable gate array (FPGA) can be reconfigured to accelerate any kernel. Due to the time cost of reconfiguration, however, FPGA cannot change the configuration frequently to execute different kernels. When mapping an application, FPGA usually needs to map all the target kernels spatially, which is limited by the area and cannot do a spatio-temporal mapping. On the contrary, CGRA can reconfigure the PE and on-chip network per cycle, thus leading to a spatio-temporal mapping.

Partitioning between CPU and CGRA
An application can have multiple kernels. Some kernels can significantly benefit from CGRA because of adequate instruction-level parallelism, while other kernels may not. With limited on-chip memory, offloading all the kernels to a tiny CGRA might be not a good choice as it might need to use main memory to store intermediate data. Lee et al. (2015) explore how to execute the whole application onto CGRA and host processor. This work first profiles the execution time and memory requirement of each kernel. Then it uses

(continued)

integer linear programming (ILP) to select which kernel to execute on CGRA and which kernel to execute on host processor. Through this method, it can maximize the utilization of CGRA and reduce the data transfer between host processor and CGRA. CGRA can be reconfigured to execute the selected kernels. However, this work only focuses on a small 4×4 CGRA and did not explore how to run multiple kernels concurrently on CGRA.

Synchronous Dataflow (SDF)

Synchronous Dataflow (SDF) is a suitable representation for application-level mapping on CGRAs. The SDF has several actors, and data are encapsulated in an object called a token. Each actor either consumes data tokens or produces tokens or both in each invocation. An actor in an SDF can be a sequential code, a single loop, or multiple loops, or any combination of them. Figure 15 shows a SDF example that has three actors: A, B, and C. Each invocation of A produces 20 tokens. Each invocation of B consumes 10 tokens from A and produces 20 tokens. Thus the SDF needs a schedule that can balance the execution of actors. $A(BC^2)^2$ and AB^2C^4 are some of the many possible bounded-buffer schedules for our example, where $A(BC^2)^2$ indicates the execution order $ABCCBCC$ and AB^2C^4 represents $ABBCCCC$. These schedules trade off between buffer requirement and throughput.

Figure 16 shows the difference between spatial and spatio-temporal mapping. In FPGA, these actors are placed spatially and scheduled to respect to the data dependency among the actors. Each actor occupies a region throughout. However, this method cannot utilize the advantage of the spatio-temporal mapping for CGRA. Specially, the SDF needs a schedule to orchestrate the actors to satisfy the data dependencies. It is hard to achieve balanced execution of the actors under the SDF schedule constraints with spatial-only mapping. On the other side, the spatio-temporal mapping of the actors provides a 3D space for the schedule that has more flexibility to map these actors.

ChordMap (Li et al. 2021) explore the mapping of the SDF onto CGRA for high throughput, spatio-temporal mapping. Given a limited scratchpad memory for the buffers, ChordMap uses a divide-and-conquer approach to partition the SDF and CGRA to reduce the complexity. ChordMap maps each sub-SDF onto corresponding sub-CGRA and uses an iterative approach

(continued)

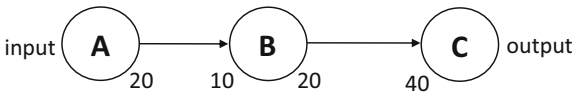


Fig. 15 SDF example

to improve the overall mapping. ChordMap can exploit the instruction-level parallelism inside actor, the parallelism among actors and their instances, and the pipeline parallelism among sub-SDFs.

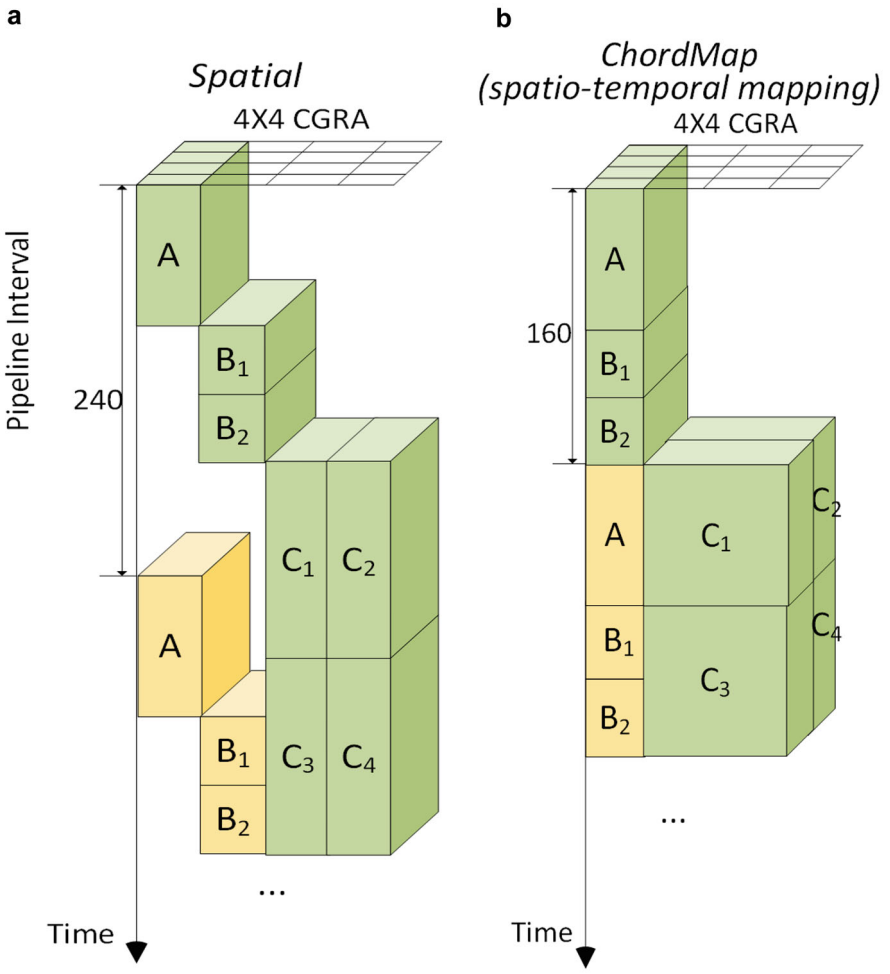


Fig. 16 Comparison between a Spatial and Spatio-Temporal mapping. (**a**) Spatial mapping. (**b**) Spatio-temporal mapping

Handling Loops with Control Flow

The statically scheduled CGRAs rely on predication to handle the loops with complex control flow (Han et al. 2013). Predication effectively translates the control flow instructions with dataflow instructions. The compiler maps both paths of each conditional branch onto the CGRA, but the instructions from the taken path are permitted to execute at runtime. This leads to resource underutilization due to static allocation of duplicate resources that are unused at runtime. A recent work 4D-CGRA (Karunaratne et al. 2019) proposes a new execution paradigm to handle control divergence at low overhead. 4D-CGRA architecture follows a semi-triggered execution model, a hybrid between sequential execution and triggered execution to accelerate loops with complex control flows. 4D-CGRA compiler places multiple shards of instructions (a portion of a basic block) from mutually exclusive execution paths on a PE and triggers a specific shard at runtime.

Scalable CGRA Mapping

Most CGRA mapping approaches are not scalable, i.e., they fail to generate high-quality mappings within an acceptable compilation time for larger CGRAs and complex application kernels. Operation placement and routing become increasingly difficult in larger CGRAs due to limited routing resources and complicated data dependencies in bigger kernels. Therefore, most CGRA mappers are only evaluated on small benchmark kernels and small CGRA sizes. Table 2 shows the DFG size, CGRA size, and compilation time of prominent CGRA mappers. SPR (Friedman et al. 2009) is the most scalable compiler evaluated on benchmark kernels with an average of 263 nodes and 16 × 16 CGRA.

Panorama is a fast and scalable mapper that generates quality mapping for complex dataflow graphs onto larger CGRA using a divide-and-conquer approach (Wijerathne et al. 2022b). It is a portable solution that can be combined with existing low-level CGRA mappers to achieve enhanced performance in a shorter compilation time. Panorama implements a high-level mapping step that finds clusters of nodes in the dataflow graph and performs cluster-level mapping to place closely related clusters on nearby CGRA PE clusters. The higher-level mapping guides the lower-level mapping, reducing overall complexity. HiMap is another fast and scalable mapping technique, although it is only specialized for

Table 2 Summary of CGRA mappers

	DFG nodes	CGRA size	Compilation time
CGRA-ME (Chin et al. 2017)	12	4	NA
SPKM (Yoon et al. 2009)	16	4 × 4	∼1 s
G-Minor (Chen and Mitra 2014)	35	4 × 4, 16 × 16	0.2 s, 7 s
EPIMAP (Hamzeh et al. 2012)	35	4 × 4, 16 × 16	54 s, 23 min
DRESC (Mei et al. 2002)	56	4 × 4	∼15 min
EMS (Park et al. 2008b)	4 ∼ 142	4 × 4	∼37 min
SPR (Friedman et al. 2009)	263	16 × 16	NA

mapping regular highly parallel kernels, as mentioned in the nested loop mapping section (Wijerathne et al. 2021a,b). Similar approaches of exploring multi-level parallelism have been studied in the context of FPGAs (Zhong et al. 2014, 2016, 2017).

Conclusions

Coarse-grained Reconfigurable Array (CGRA) has emerged as a popular, general-purpose, spatial accelerator that can support high-performance, energy efficiency, and flexibility across multiple application domains. In this chapter, we presented an overview of the architectural and compilation innovations over the last two decades to better realize the potential of the CGRAs. There remain multiple challenges and opportunities in this space, including but not limited to, scalability for more complex applications, memory management, runtime power management, and specializations for important and emerging application domains.

Acknowledgments This work is partially supported by the National Research Foundation, Singapore, under its Competitive Research Programme Award NRF-CRP23-2019-0003.

References

Ahn M, Yoon JW, Paek Y, Kim Y, Kiemb M, Choi K (2006) A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures. In: Proceedings of the Conference on Design, Automation and Test in Europe: Proceedings. European Design and Automation Association, pp 363–368

Alle M, Varadarajan K, Ramesh RC, Nimmy J, Fell A, Rao A, Nandy S, Narayan R (2008) Synthesis of application accelerators on runtime reconfigurable hardware. In: 2008 International Conference on Application-Specific Systems, Architectures and Processors. IEEE, Munich, Germany, pp 13–180

Amdahl GM (1967) Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the Spring Joint Computer Conference, 18–20 Apr, 1967, pp 483–485

Bandara TK, Wijerathne D, Mitra T, Peh LS (2022) REVAMP: a systematic framework for heterogeneous CGRA realization. In: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). ACM, Lausanne, Switzerland

Bansal N, Gupta S, Dutt N, Nicolau A (2003) Analysis of the performance of coarse-grain reconfigurable architectures with different processing element configurations. Proc. of Workshop on Application Specific Processors, vol. 12

Baumgarte V, Ehlers G, May F, Nückel A, Vorbach M, Weinhardt M (2003) PACT XPP – a self-reconfigurable data processing architecture. J Supercomput 26(2):167–184

Betz V, Rose J (1997) VPR: a new packing, placement and routing tool for FPGA research. In: International Workshop on Field Programmable Logic and Applications. Springer, Berlin Heidelberg, 1, pp 213–222

Brenner JA, Fekete SP, Van Der Veen JC (2009) A minimization version of a directed subgraph homeomorphism problem. Math Methods Oper Res 69(2):281–296

Burns GF, Jacobs M, Lindwer M, Vandewiele B (2004) Exploiting parallelism, while managing complexity using Silicon Hive programming tools. White paper vol. 42, p. 43, 2004.
Cao P, Liu B, Yang J, Yang J, Zhang M, Shi L (2017) Context management scheme optimization of coarse-grained reconfigurable architecture for multimedia applications. IEEE Trans Very Large Scale Integr (VLSI) Syst 17, 2321–2331
Carballo J-A, Chan W-TJ , Gargini PA, Kahng AB, Nath S (2014) ITRS 2.0: toward a re-framing of the semiconductor technology roadmap. In: 2014 IEEE 32nd International Conference on Computer Design (ICCD). IEEE, pp 139–146
Chaudhuri S, Hetzel A (2017) SAT-based compilation to a non-vonNeumann processor. In: Proceedings of the 36th International Conference on Computer-Aided Design. IEEE Press, Irvine, CA, USA, pp 675–682
Chen L, Mitra T (2014) Graph minor approach for application mapping on CGRAs. ACM Trans Reconfig Technol Syst (TRETS) 7(3):1–25
Chen Y-H, Yang T-J, Emer J, Sze V (2019) Eyeriss v2: a flexible accelerator for emerging deep neural networks on mobile devices. IEEE J Emerg Sel Top Circuits Syst 9(2):292–308
Chin SA, Anderson JH (2018) An architecture-agnostic integer linear programming approach to CGRA mapping. In: 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC). IEEE, pp 1–6
Chin SA, Sakamoto N, Rui A, Zhao J, Kim JH, Hara-Azumi Y, Anderson J (2017) CGRA-ME: a unified framework for CGRA modelling and exploration. In: 2017 IEEE 28th International Conference on Application-Specific Systems, Architectures and Processors (ASAP). IEEE, Seattle, WA, USA, pp 184–189
Choi K (2011) Coarse-grained reconfigurable array: architecture and application mapping. IPSJ Trans Syst LSI Des Methodol 4:31–46
Compton K, Hauck S (2002) Reconfigurable computing: a survey of systems and software. ACM Comput Surv (csuR) 34(2):171–210
Dally WJ, Turakhia Y, Han S (2020) Domain-specific hardware accelerators. Commun ACM 63(7):48–57
DARPA software defined hardware (2019). Online. Available: https://www.darpa.mil/program/software-defined-hardware
Dave S, Balasubramanian M, Shrivastava A (2018) RAMP: resource-aware mapping for CGRAs. In: 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC). IEEE, San Francisco, CA, USA, pp 1–6
Dennard RH, Gaensslen FH, Yu H-N, Rideout VL, Bassous E, LeBlanc AR (1974) Design of ion-implanted MOSFET's with very small physical dimensions. IEEE J Solid-State Circuits 9(5):256–268
De Sutter B, Coene P, Vander Aa T, Mei B (2008) Placement-and-routing-based register allocation for coarse-grained reconfigurable arrays. In: Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers and Tools for Embedded System, ser. LCTES'08. ACM, Tucson, Arizona, USA, pp 151–160
Di Battista G, Patrignani M, Vargiu F (1998) A split&push approach to 3D orthogonal drawing. In: International Symposium on Graph Drawing. Springer, Berlin, Heidelberg, pp 87–101
Eisenbeis C, Lelait S, Marmol B (1995) The meeting graph: a new model for loop cyclic register allocation. In: Proceedings of the 1995 International Federation for Information Processing Working Group, pp 264–267
Emani M, Vishwanath V, Adams C, Papka ME, Stevens R, Florescu L, Jairath S, Liu W, Nama T, Sujeeth A (2021) Accelerating scientific applications with SambaNova reconfigurable dataflow architecture. Comput Sci Eng 23(2):114–119
Fleming KE, Glossop KD, Steely SC Jr, Tang J, Gara AG et al (2020) Processors, methods, and systems with a configurable spatial accelerator. US Patent 10,558,575, 11 Feb 2020
Fortune S, Hopcroft J, Wyllie J (1980) The directed subgraph homeomorphism problem. Theor Comput Sci 10(2):111–121

Friedman S, Carroll A, Van Essen B, Ylvisaker B, Ebeling C, Hauck S (2009) SPR: an architecture-adaptive CGRA mapping tool. In: Proceedings of the 17th ACM/SIGDA International Symposium on Field Programmable Gate Arrays, ser. FPGA'09. ACM, pp 191–200

Fujii T, Toi T, Tanaka T, Togawa K, Kitaoka T, Nishino K, Nakamura N, Nakahara H, Motomura M (2018) New generation dynamically reconfigurable processor technology for accelerating embedded AI applications. In: 2018 IEEE Symposium on VLSI Circuits. IEEE, Honolulu, HI, USA, pp 41–42

Gao M, Kozyrakis C (2016) HRL: efficient and flexible reconfigurable logic for near-data processing. In: 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, Barcelona, spain, pp 126–137

Ghorpade J, Parande J, Kulkarni M, Bawaskar A (2012) GPGPU processing in CUDA architecture. arXiv preprint arXiv:1202.4347

Hameed R, Qadeer W, Wachs M, Azizi O, Solomatnikov A, Lee BC, Richardson S, Kozyrakis C, Horowitz M (2010) Understanding sources of inefficiency in general-purpose chips. In Proceedings of the 37th Annual International Symposium on Computer Architecture, pp 37–47

Hamzeh M, Shrivastava A, Vrudhula S (2012) EPIMap: using epimorphism to map applications on CGRAs. In: Proceedings of the 49th Annual Design Automation Conference, pp 1284–1291

Hamzeh M, Shrivastava A, Vrudhula S (2013) REGIMap: register-aware application mapping on coarse-grained reconfigurable architectures (CGRAs). In: Proceedings of the 50th Annual Design Automation Conference, pp 1–10

Han K, Ahn J, Choi K (2013) Power-efficient predication techniques for acceleration of control flow execution on CGRA. ACM Trans Architecture Code Optim (TACO) 10(2):1–25

Hatanaka A, Bagherzadeh N (2007) A modulo scheduling algorithm for a coarse-grain reconfigurable array template. In: Proceedings of the 21st International Parallel and Distributed Processing Symposium, ser. IPDPS'07. IEEE, Long Beach, CA, USA, pp 1–8

Hennessy JL, Patterson DA (2011) Computer architecture: a quantitative approach. Elsevier, Amsterdam

Jafri SMAH, Tajammul MA, Hemani A, Paul K, Plosila J, Ellervee P, Tenuhnen H (2015) Polymorphic configuration architecture for CGRAs. IEEE Trans Very Large Scale Integr (VLSI) Syst 24(1):403–407

Jouppi NP, Young C, Patil N, Patterson D, Agrawal G, Bajwa R, Bates S, Bhatia S, Boden N, Borchers A et al (2017) In-datacenter performance analysis of a tensor processing unit. In: Proceedings of the 44th Annual International Symposium on Computer Architecture, pp 1–12

Kågström B, Ling P, Van Loan C (1998) GEMM-based level 3 BLAS: high-performance model implementations and performance evaluation benchmark. ACM Trans Math Softw (TOMS) 24(3):268–302

Karunaratne M, Mohite AK, Mitra T, Peh L-S (2017) HyCUBE: a CGRA with reconfigurable single-cycle multi-hop interconnect. In: Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE. IEEE, Austin, TX, USA, pp 1–6

Karunaratne M, Tan C, Kulkarni A, Mitra T, Peh L-S (2018) DNestMap: mapping deeply-nested loops on ultra-low power CGRAs. In: 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC). IEEE, San Francisco, CA, USA, pp 1–6

Karunaratne M, Wijerathne D, Mitra T, Peh L-S (2019) 4D-CGRA: introducing branch dimension to spatio-temporal application mapping on CGRAs. In: 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). IEEE, Westminster, CO, USA, pp 1–8

Kim Y, Lee J, Shrivastava A, Yoon J, Paek Y (2010) Memory-aware application mapping on coarse-grained reconfigurable arrays. In: International Conference on High-Performance Embedded Architectures and Compilers. Springer, pp 171–185

Kim Y, Lee J, Shrivastava A, Paek Y (2010) Operation and data mapping for CGRAs with multi-bank memory. ACM SIGPLAN Not 45(4):17–26

Kirkpatrick S, Gelatt CD, Vecchi MP (1983) Optimization by simulated annealing. Science 220(4598):671–680

Kuon I, Rose J (2007) Measuring the gap between FPGAs and ASICs. IEEE Trans Comput-Aided Des Integr Circuits Syst 26(2):203–215

Kuon I, Tessier R, Rose J (2008) FPGA architecture: survey and challenges. Now Publishers Inc., Hanover, MA 02339 USA

Kwong J, Chandrakasan AP (2011) An energy-efficient biomedical signal processing platform. IEEE J Solid-State Circuits 46(7):1742–1753

Lee P, Kedem ZM (1990) Mapping nested loop algorithms into multidimensional systolic arrays. IEEE Trans Parallel Distrib Syst 1(1):64–76

Lee G, Choi K, Dutt ND (2011) Mapping multi-domain applications onto coarse-grained reconfigurable architectures. IEEE Trans Comput-Aided Des Integr Circuits Syst 30(5):637–650

Lee J, Seo S, Lee H, Sim HU (2014) Flattening-based mapping of imperfect loop nests for CGRAs. In: Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis. ACM, Uttar Pradesh, India, p 9

Lee H, Nguyen D, Lee J (2015) Optimizing stream program performance on CGRA-based systems. In: Proceedings of the 52nd Annual Design Automation Conference, pp 1–6

Li S, Ebeling C (2008) QuickRoute: a fast routing algorithm for pipelined architectures. In: Proceedings on Field-Programmable Technology, 2004. IEEE International Conference. IEEE, Brisbane, NSW, Australia, pp 73–80

Li Z, Wijerathne D, Chen X, Pathania A, Mitra T (2021) ChordMap: automated mapping of streaming applications onto CGRA. IEEE Trans Comput-Aided Des Integr Circuits Syst 41(2):306–319

Li Z, Wu D, Wijerathne D, Mitra T (2022) LISA: graph neural network based portable mapping on spatial accelerators. In: 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA). IEEE, Seoul, Korea (South)

Liu D, Yin S, Liu L, Wei S (2013) Polyhedral model based mapping optimization of loop nests for CGRAs. In: Proceedings of the 50th Annual Design Automation Conference. ACM, San Francisco, CA, USA, p 19

Liu D, Yin S, Luo G, Shang J, Liu L, Wei S, Feng Y, Zhou S (2018) Data-flow graph mapping optimization for CGRA with deep reinforcement learning. IEEE Trans Comput-Aided Des Integr Circuits Syst 38(12):2271–2283

Liu L, Zhu J, Li Z, Lu Y, Deng Y, Han J, Yin S, Wei S (2019) A survey of coarse-grained reconfigurable architecture and design: taxonomy, challenges, and applications. ACM Comput Surv (CSUR) 52(6):1–39

Lu W, Yan G, Li J, Gong S, Han Y, Li X (2017) FlexFlow: a flexible dataflow accelerator architecture for convolutional neural networks. In: 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, Austin, TX, USA, pp 553–564

McMurchie L, Ebeling C (2008) PathFinder: a negotiation-based performance-driven router for FPGAs. In: Reconfigurable computing. Elsevier, Burlington, Massachusetts, pp 365–381

Mei B, Vernalde S, Verkest D, De Man H, Lauwereins R (2002) DRESC: a retargetable compiler for coarse-grained reconfigurable architectures. In: 2002 IEEE International Conference on Field-Programmable Technology, 2002 (FPT). Proceedings. IEEE, Hong Kong, China, pp 166–173

Mei B, Vernalde S, Verkest D, De Man H, Lauwereins R (2003a) ADRES: an architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In: Proceedings of the 13th International Conference on Field Programmable Logic and Application, ser. FPL'03. Springer, Berlin Heidelberg, pp 61–70

Mei B, Vernalde S, Verkest D, De Man H, Lauwereins R (2003b) Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In: Proceedings of the 2003 Conference on Design, Automation and Test in Europe, ser. DATE'03. IEEE, Munich, Germany, pp 296–301

Mitra T (2015) Heterogeneous multi-core architectures. Inf Media Technol 10(3):383–394

Moore GE et al (1998) Cramming more components onto integrated circuits. Proceedings of the IEEE 86(1): 82–85

Nicol C (2017) A coarse grain reconfigurable array (CGRA) for statically scheduled data flow computing. Wave Computing White Paper

Nowatzki T, Gangadhar V, Ardalani N, Sankaralingam K (2017) Stream-dataflow acceleration. In: 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA). IEEE, Toronto, ON, Canada, pp 416–429

Park H, Fan K, Mahlke SA, Oh T, Kim H, Kim H-S (2008a) Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, ser. PACT'08. ACM, Toronto, Ontario, Canada, pp 166–176

Park H, Fan K, Mahlke SA, Oh T, Kim H, Kim H-S (2008b) Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, pp 166–176

Patterson DA (2006) Future of computer architecture. In: Berkeley EECS Annual Research Symposium (BEARS), College of Engineering, UC Berkeley, US

Podobas A, Sano K, Matsuoka S (2020) A survey on coarse-grained reconfigurable architectures from a performance perspective. IEEE Access 8:146719–146743

Prabhakar R, Zhang Y, Koeplinger D, Feldman M, Zhao T, Hadjis S, Pedram A, Kozyrakis C, Olukotun K (2017) Plasticine: a reconfigurable architecture for parallel patterns. In: 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA). IEEE, Toronto, ON, Canada, pp 389–402

Rashid M, Imran M, Jafri AR, Al-Somani TF (2019) Flexible architectures for cryptographic algorithms – a systematic literature review. J Circuits Syst Comput 28(03):1930003

Rau BR (1994) Iterative modulo scheduling: an algorithm for software pipelining loops. In: Proceedings of the 27th Annual International Symposium on Microarchitecture. ACM, San José, CA, USA, pp 63–74

Robertson N, Seymour PD (1990) Graph minors. IX. Disjoint crossed paths. J Comb Theory Ser B 49(1):40–77

Shao YS, Reagen B, Wei G-Y, Brooks D (2015) The Aladdin approach to accelerator design and modeling. IEEE Micro 35(3):58–70

Singh H, Lee MH, Lu G, Kurdahi FJ, Bagherzadeh N, Chaves Filho EM (2000) MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications. IEEE Trans Comput 49(5):465–481

Singh H, Lee M-H, Lu G, Kurdahi FJ, Bagherzadeh N, Chaves Filho EM (2000) MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications. IEEE Trans Comput 49(5):465–481

Suh D, Kwon K, Kim S, Ryu S, Kim J (2012) Design space exploration and implementation of a high performance and low area coarse grained reconfigurable processor. In: 2012 International Conference on Field-Programmable Technology. IEEE, Seoul, Korea (South), pp 67–70

Tu F, Yin S, Ouyang P, Tang S, Liu L, Wei S (2017) Deep convolutional neural network architecture with reconfigurable computation patterns. IEEE Trans Very Large Scale Integr (VLSI) Syst 25(8):2220–2233

Tuhin MAA, Norvell TS (2008) Compiling parallel applications to coarse-grained reconfigurable architectures. In: 2008 Canadian Conference on Electrical and Computer Engineering. IEEE, Niagara Falls, ON, Canada, pp 001723–001728

Venkataramani S, Choi J, Srinivasan V, Wang W, Zhang J, Schaal M, Serrano MJ, Ishizaki K, Inoue H, Ogawa E et al (2019) DeepTools: compiler and execution runtime extensions for rapid AI accelerator. IEEE Micro 39(5):102–111

Wang Y, Li P, Zhang P, Zhang C, Cong J (2013) Memory partitioning for multidimensional arrays in high-level synthesis. In: Proceedings of the 50th Annual Design Automation Conference, pp 1–8

Wang B, Karunarathne M, Kulkarni A, Mitra T, Peh L-S (2019) HyCUBE: a 0.9 V 26.4 MOPS/mW, 290 pJ/op, power efficient accelerator for IoT applications. In: 2019 IEEE Asian Solid-State Circuits Conference (A-SSCC). IEEE, Austin, TX, USA, pp 133–136

Wijerathne D, Li Z, Karunarathne M, Pathania A, Mitra T (2019) Cascade: high throughput data streaming via decoupled access-execute CGRA. ACM Trans Embed Comput Syst (TECS) 18(5s):1–26

Wijerathne D, Li Z, Pathania A, Mitra T, Thiele L (2021a) HiMap: fast and scalable high-quality mapping on CGRA via hierarchical abstraction. IEEE Trans Comput-Aided Des Integr Circuits Syst 41(10):3290–3303

Wijerathne D, Li Z, Pathania A, Mitra T, Thiele L (2021b) HiMap: fast and scalable high-quality mapping on CGRA via hierarchical abstraction. pp 1192–1197

Wijerathne D, Li Z, Karunaratne M, Peh L-S, Mitra T (2022a) Morpher: an open-source integrated compilation and simulation framework for CGRA. In: Workshop on Open-Source EDA Technology (WOSET)

Wijerathne D, Li Z, Bandara TK, Mitra T (2022b) PANORAMA: divide-and-conquer approach for mapping complex loop kernels on CGRA. In: 2022 59th ACM/EDAC/IEEE Design Automation Conference (DAC). IEEE, San Francisco, CA, USA, pp 1–6

Yin S, Yao X, Liu D, Liu L, Wei S (2015) Memory-aware loop mapping on coarse-grained reconfigurable architectures. IEEE Trans Very Large Scale Integr (VLSI) Syst 24(5):1895–1908

Yin s, Yao x, Lu T, Liu L, Wei S (2016a) Joint loop mapping and data placement for coarse-grained reconfigurable architecture with multi-bank memory. In: Proceedings of the 35th International Conference on Computer-Aided Design, pp 1–8

Yin S, Lin X, Liu L, Wei S (2016b) Exploiting parallelism of imperfect nested loops on coarse-grained reconfigurable architectures. IEEE Trans Parallel Distrib Syst 27(11):3199–3213

Yin S, Liu D, Sun L, Liu L, Wei S (2017a) DFGNet: mapping dataflow graph onto CGRA by a deep learning approach. In: 2017 IEEE International Symposium on Circuits and Systems (ISCAS). IEEE, Baltimore, MD, USA, pp 1–4

Yin S, Yao X, Lu T, Liu D, Gu J, Liu L, Wei S (2017b)Conflict-free loop mapping for coarse-grained reconfigurable architecture with multi-bank memory. IEEE Trans Parallel Distrib Syst 28(9):2471–2485

Yoo J, Yan L, El-Damak D, Altaf MAB, Shoeb AH, Chandrakasan AP (2012) An 8-channel scalable EEG acquisition SoC with patient-specific seizure classification and recording processor. IEEE J Solid-State Circuits 48(1):214–228

Yoon JW, Shrivastava A, Park S, Ahn M, Paek Y (2009) A graph drawing based spatial mapping algorithm for coarse-grained reconfigurable architectures. IEEE Trans Very Large Scale Integr (VLSI) Syst 17(11):1565–1578

Zalamea J, Llosa J, Ayguadé E, Valero M (2001) MIRS: modulo scheduling with integrated register spilling. In: International Workshop on Languages and Compilers for Parallel Computing. Springer, Berlin, Heidelberg, pp 239–253

Zhong G, Venkataramani V, Liang Y, Mitra T, Niar S (2014) Design space exploration of multiple loops on FPGAs using high level synthesis. In: 2014 IEEE 32nd International Conference on Computer Design (ICCD). IEEE, Seoul, Korea (South), pp 456–463

Zhong G, Prakash A, Liang Y, Mitra T, Niar S (2016) Lin-Analyzer: a high-level performance analysis tool for FPGA-based accelerators. In: 53rd ACM/EDAC/IEEE Design Automation Conference (DAC). IEEE, San Francisco, CA, USA, pp 1–6

Zhong G, Prakash A, Wang S, Liang Y, Mitra T, Niar S (2017) Design space exploration of FPGA-based accelerators with multi-level parallelism. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017. IEEE, Dresden, Germany, pp 1141–1146

Dynamic and Partial Reconfiguration of FPGAs

15

Suhaib A. Fahmy and Krishnan B. Iyer

Contents

Introduction 508
FPGA Configuration 510
Designing Partially Reconfigurable Systems 512
Managing Partial Reconfiguration 517
Applications of Dynamic Partial Reconfiguration 519
  Computing Infrastructure and Virtualization 520
  Design Compilation 521
  Adaptive Systems 522
  Machine Learning 524
  Reliability and Harsh Environments 524
Research Directions 525
Conclusions 526
References 526

Abstract

The reconfigurability of FPGAs is a unique capability that can be exploited beyond just repurposing or modifying hardware designs. Static reconfiguration, where a single monolithic hardware design is replaced by another, allows for in-field upgradability and enhancements, de-risks hardware deployment, and enables their function as off-the-shelf programmable devices. However, this

S. A. Fahmy (✉)
King Abdullah University of Science and Technology (KAUST), Department of Computer, Electrical and Mathematical Sciences and Engineering, Thuwal, Saudi Arabia
e-mail: suhaib.fahmy@kaust.edu.sa

K. B. Iyer
Computer Science, King Abdullah University of Science and Technology, Thuwal, Saudi Arabia
e-mail: krishnan.iyer@kaust.edu.sa

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_51

configurability, through modification of configuration memory contents, also opens the door to dynamic reconfiguration, where hardware designs are changed at runtime to serve different purposes. More advanced still is the ability to modify only portions of the hardware architecture, while the rest remains functional. By closing the loop, wherein the static part of the hardware is responsible for controlling the reconfiguration of the dynamic part, self-reconfiguring systems are possible. This chapter explores the dynamic and partial reconfiguration capabilities of FPGAs from the perspectives of the architecture, the programming model, and the applications that can leverage these unique capabilities.

Keywords

Reconfigurable computing · Field programmable gate arrays · Partial reconfiguration · Self-adaptive hardware

Introduction

The flexibility of field-programmable gate arrays (FPGAs) is afforded by the various programmable elements discussed in Boutros and Betz (2023). The configuration memory is tightly coupled with these elements, with individual bits of the configuration memory controlling features of the various elements to implement supported functions. These can be the contents of the look-up tables (LUTs) to implement logic functions, whether logic element flip-flops are enabled or not and how they are connected, whether the LUT implements a fractured function, which input and output ports of a switch box to connect, single-ended vs. differential I/O configurations, and the operating modes of hardened macros like DSP Blocks and Block Memories. Figures 1 and 2 show a representation of how configuration affects logic elements and routing.

The contents of the whole configuration memory represent the hardware context of the FPGA, that is, the way that every programmable element is configured to implement the required circuit. This binary data is referred to as the *bitstream*. Loading a bitstream into the FPGA sets up the FPGA as determined by the way the contents configure each of the millions of programmable elements. It is worth noting that it might be possible to load a bitstream into the FPGA that configures a circuit that does not function correctly or even worse causes physical damage to the FPGA, hence the need for approaches to ensure only valid bitstreams are loaded, as will be discussed later.

The most widely used FPGAs today have a volatile SRAM-based configuration memory, thereby enabling reconfiguration: the rewriting of the configuration bitstream. While early devices required this to be done while in a reset state—a *static* reconfiguration—many devices today allow *dynamic* reconfiguration, that is, the rewriting of the bitstream while the device is active, effectively a hardware context switch. A further enhancement that significantly increases the utility of this feature is *partial* reconfiguration, wherein only portions of the configuration memory are replaced, thereby modifying the context of only the corresponding areas of the

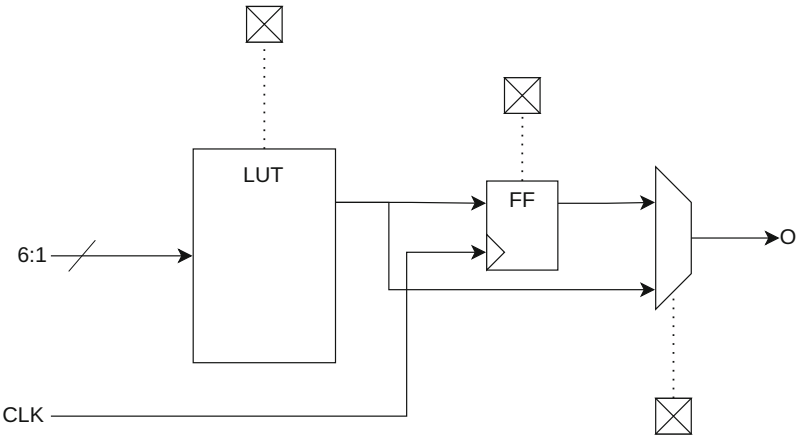


Fig. 1 Configuration of a simplified logic element showing 64-bit LUT6 contents determining logic function, flip-flop as latch with initial state, and multiplexer select pins set by the values in the configuration memory

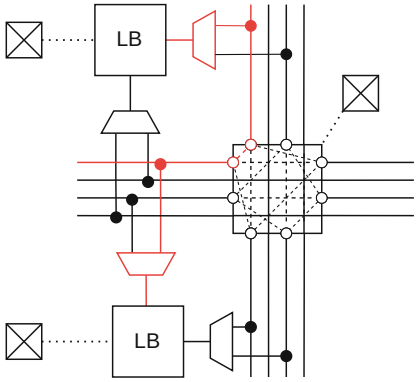


Fig. 2 Configuration of a route between two logic blocks through configuring the connection boxes and the switch box to connect them

device. This opens the door to hardware designs where parts can be modified at runtime based on functional needs. A *static region* contains the parts of the hardware design that remain active throughout operation, while one or more *partial regions* contain functionality which can be swapped out during operation.

This capability has found use in a variety of applications, such as cognitive radio systems that modify their hardware baseband processing depending on operating requirements, computer vision systems that adapt the type of image processing depending on the properties of the scene, virtualization of FPGA interfaces for generalized accelerator deployments, and resilient designs that are robust to radiation effects in space.

This chapter discusses the mechanisms that enable dynamic and partial reconfiguration and how to design such systems and gives examples of applications that exploit these capabilities. Future challenges in this research area are also identified. Various existing surveys delve into these aspects in more detail and the authors

are referred to these (Compton and Hauck (2002), Koch (2012), Vipin and Fahmy (2018), and Vaishnav et al. (2018)).

FPGA Configuration

The configuration memory is what enables the flexibility of FPGAs to be exploited. Different memory technologies are used to implement FPGA configuration memories (Kuon et al. 2008); among them Flash, Antifuse, and SRAM are prevalent. Antifuse memory is one-time writable and hence FPGAs that use it do not provide features like in-system programming and reconfiguration. SRAM memory is typically latch-based, which provides stability against minor noise or glitches, and offers write stability due to the feedback in the cells. However, due to volatility, SRAM-based FPGAs must be configured each time they are powered on. Flash-based configuration memory is non-volatile, thereby offering instant power-up, but suffers from limited write cycles, and is therefore not ideally suited for reconfigurable applications. Hence, SRAM-based FPGAs typically offer the reconfiguration capability that is discussed in this chapter. However, due to their volatility, bitstreams must be stored in non-volatile memory, e.g., Flash, to be loaded onto the FPGA after any power reset.

One complication with storing bitstreams in external non-volatile memory is the potential for a bitstream to be read out by a malicious attacker who can then use it on their own (identical) device to implement the same circuit. It is worth clarifying that while this is problematic in the general sense, reverse engineering the bitstream to obtain the original hardware description language source code for design remains as complex as reverse engineering a software binary to obtain its source code. Rather the bitstream can be used to extract a complex low-level netlist of device primitives that requires significant effort to analyze (Benz et al. 2012).

However, considering these designs represent the value offered by FPGA system designers, there are various mechanisms by which bitstreams are secured. Both Intel/Altera and AMD/Xilinx support on-chip decryption of encrypted bitstreams (Xilinx 2024; Karen Horovitz 2024; Intel 2021), with a dedicated memory for storing the encryption key. At present, they both use AES encryption, while AMD/Xilinx uses RSA and Intel/Altera uses ECDSA for authentication. This allows bitstreams to be locked to writing a specific device. To make encryption effective, vendors disable readback of the keys from memory. Some low-end FPGAs do not include these features, instead relying on obfuscation and the proprietary nature of the bitstream format. Various academic projects have tried to address this challenge through custom encryption and authentication schemes (Vliegen et al. 2013; Duncan et al. 2019). A thorough review of the security implications of FPGA reconfigurability is presented in Proulx et al. (2023).

Bitstreams can also be compressed to allow faster reconfiguration. Custom schemes would still require decompression before being passed to the configuration interface, so they only offer the benefit of reduced bitstream size in memory.

AMD/Xilinx supports a proprietary compression technique enabled by setting a special flag during bitstream generation and this scheme is natively supported in the hardware. This offers high bitstream compression when there is lower device utilization (Xilinx 2023). A custom scheme that exploits similarity in bitstream data based on a custom relocation scheme is presented in Beckhoff et al. (2014).

The bitstream is not itself a one-to-one mapping to the contents of the configuration memory but rather a microcode for the FPGA configuration controller to encode and direct data to specific locations in the configuration memory. Being proprietary, it requires significant effort to fully decode this relationship, though some understanding of the frame structure and bitstream format can be gained from the AMD/Xilinx official configuration guides (Xilinx 2023). Various academic efforts have successfully reverse engineered these bitstream formats for some devices to enable open-source tools to target them (Soni et al. 2013).

FPGAs can be configured by writing bitstreams over a variety of interfaces. The most basic approach is over the JTAG debugging interface. This is slow as it is not optimized for throughput and bitstreams are sent as individual memory writes. External configuration interfaces are faster but must be managed by a distinct processor. FPGA SoCs have internal configuration interfaces that the processors can manage to configure the FPGA portions on the same die. Interface selection determines where the bitstreams can be stored and how long the reconfiguration process takes. This is discussed in more detail in Section "Managing Partial Reconfiguration".

Generally, reconfiguring an FPGA required a reset to be applied first, to protect against stray values which might cause unexpected behavior after configuration. More recent FPGAs allow dynamic reconfiguration, allowing a new bitstream to be written while the device is active. This enables a new form of reconfigurability: partial reconfiguration (PR).

PR allows reconfiguration of a specific region of the FPGA while the rest remains operational. To exploit PR, a design must be partitioned into a static region (SR) and one or more partially reconfigurable regions (PRRs). The static region maintains its hardware context throughout the operational lifetime of the device (until power off for devices with volatile configuration memory). It houses any fixed functionality required throughout operation, such as external interface logic, a soft processor that manages the system, memory connections, and other peripherals. Partially reconfigurable regions have multiple contexts that can be swapped at runtime. These must be arranged on the device according the various constraints to be valid. To swap contexts, a partial bitstream must be loaded to reconfigure the relevant region.

PR presents some design challenges discussed in Section "Designing Partially Reconfigurable Systems" such as the coarse granularity of PR, relocation of bitstreams, the consistency model for reconfiguration, decoupling of signals during reconfiguration, high-level design support, and simulation support. The consideration impacts build and runtime development flow significantly. For example, simulation support is lacking in vendor tools, and hence, potential pitfalls caused by dynamic reconfiguration can be missed.

Designing Partially Reconfigurable Systems

One of the obstacles to wider adoption of PR has been design complexity. Most FPGA designers are content with writing RTL code and letting FPGA tools manage the complexity of mapping to the target architecture. Indeed, in such cases, the designer need not know much about the types of resources on the FPGA and their arrangement in silicon. Optimization for high frequency tends to require both some understanding of the underlying hardware primitives and, potentially, some spatial *floorplanning*, wherein the location of hardware primitives is fixed. However, this remains a specialist skill. Designing a PR system on FPGA requires awareness of this physical arrangement and consideration of its features at a finer level than many designers consider. FPGA tool vendors have been improving their tools recently, but this additional complexity remains a barrier to entry. The PR development flows from both AMD/Xilinx and Intel/Altera follow a similar structure.

For consistency, a common terminology is defined first. PR development requires the area of the FPGA to be divided into a *static region* (SR) and one or more *partially reconfigurable regions* (PRRs). The static region hosts the static module, which is initialized once during the configuration lifetime (until power-off). The static module can include logic to manage partial reconfiguration, such as communication with the ICAP or decoupling signal circuitry.

The partially reconfiguration regions (PRRs) can be reconfigured multiple times during system lifetime. Reconfigurable modules (RMs) are the designs configured into the PRRs. A *configuration* is a valid combination of the SR and a set of RMs allocated to the PRRs in the design. A *full* bitstream includes the static module and an RM instance for each PRR. A *partial* bitstream only contains an instance of an RM for a particular PRR. Figure 3 shows these regions and the associated bitstreams. In generating partial bitstreams for RMs, these are validated against the static module to ensure correctness and compatibility. Subsequently, all bitstreams, including static and partial, must be generated prior to configuration. Since both static and partial bitstreams must be generated through the full hardware implementation flow, which is time-consuming, their functionality, arrangement, and design must be determined up front.

Signals that interface the SR and PRRs must be locked in place (which is now automated by the tools), so that the multiple RMs all route their matching interfaces to these signals. RMs in the same PRR can have different interface signals, but the SR must implement all the required signals to interface with them.

The PR design tools start by synthesizing all RMs separately. The SR is also synthesized with the RMs replaced by black boxes, i.e., empty modules. Only during the implementation stage are the synthesized netlists of the RMs populated into the black boxes.

PRRs must be drawn manually in the implementation stage, ensuring they align with clock region boundaries. It's preferred that PRRs are rectangularly shaped in order to have uniformity in resources; any other shape is likely to cause routing congestion (Xilinx 2023).

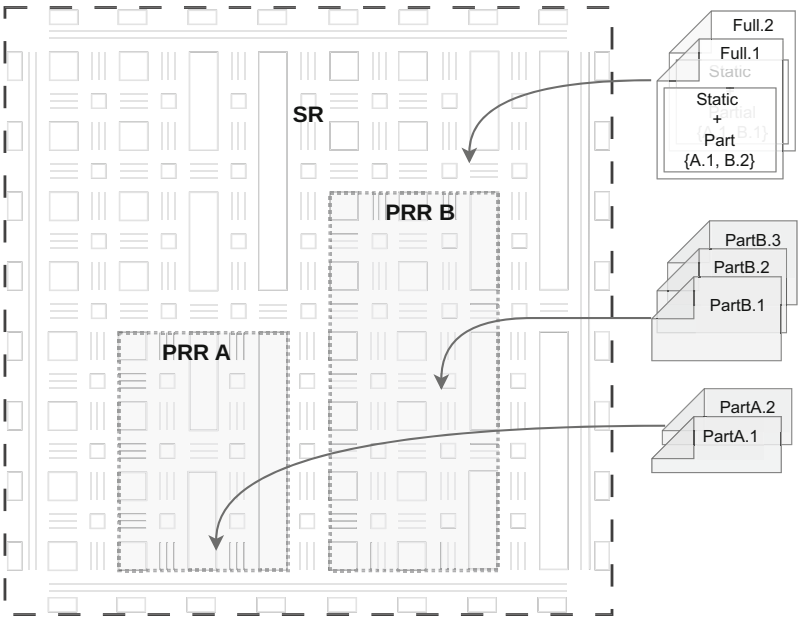


Fig. 3 A partially reconfigurable design comparing a static region (SR) and two partially reconfigurable regions (PRRs), A and B. Full bitstreams can be used to configure the whole FPGA, while partial bitstreams allow individual PRRs to be configured to a specific function. The granularity of regions is limited to device-specific constraints based on clock regions and resource arrangements

Additionally, not all resources are reconfigurable. LUTs, FFs, DSPs, CMACs, PCIe interfaces, clocks, and clocks modifying logic such as BUFG, PLL, etc., are reconfigurable. Similarly, I/O and I/O-related components such as drive strength, driver output impedance, SERDES, etc., are also reconfigurable. However, components such as BSCAN, JTAG, and ICAP are not reconfigurable.

One RM for each PRR is incorporated and a complete netlist (comprising the SR and an RM for each PRR) is generated. At this point, the static design is locked, meaning that its physical use of resources is fixed. Subsequent RMs are swapped into each PRR, and the same process repeated to determine their own implementation.

Netlists are generated for all configurations defined by the designer, resulting in a full bitstream for every configuration and partial bitstreams as needed for the PRRs to implement all configurations. Figure 4 shows an overview of the stages of designing a PR system. There are a number of academic tools that seek to further enhance the PR build flow.

Determining how many PRRs to use and which RMs to allocate to each PRR is non-trivial. Fewer PRRs containing multiple larger RMs mean that each time any part of an RM needs to be modified, the whole PRR must be reconfigured,

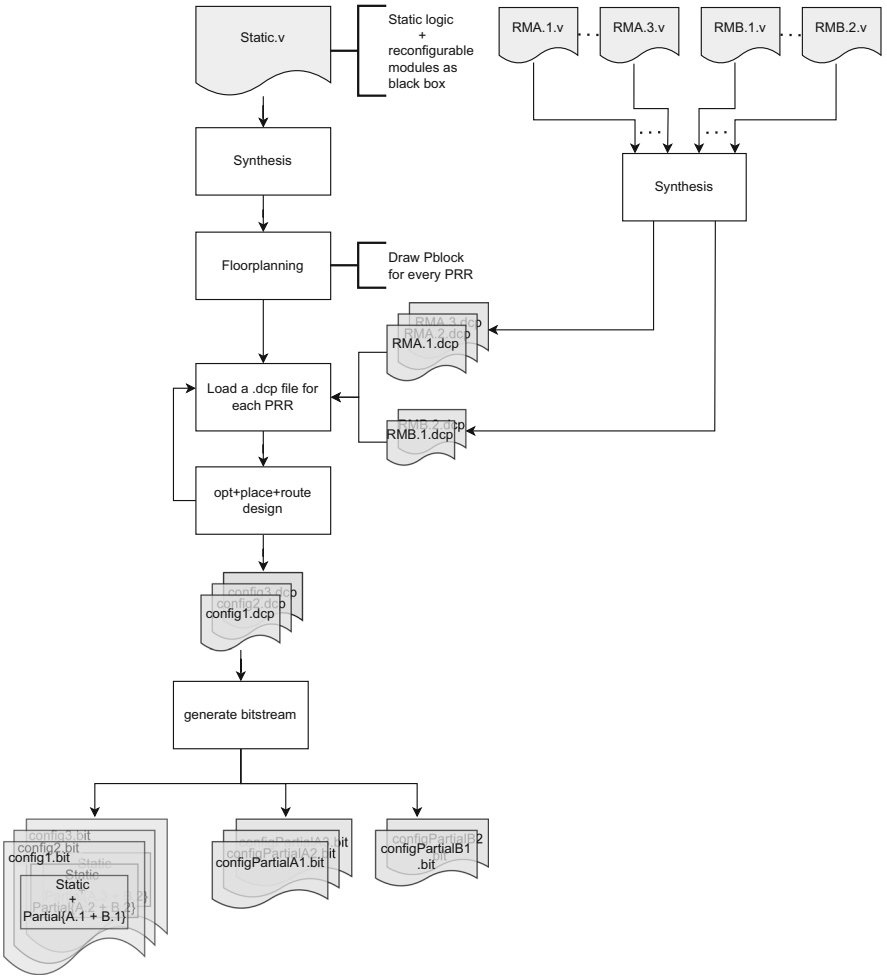


Fig. 4 The partial reconfiguration design flow. Reconfigurable modules are separately synthesized. Static logic is first passed to the design flow with reconfigurable modules as a black box. A floorplan is created for the PRRs, each of which is populated by the synthesized modules. Placement and routing are carried out on a valid configuration with PRRs populated with the corresponding RMs. This process is repeated for all valid configurations. Finally, bitstreams are generated for valid configurations

potentially resulting in some non-functional time and also slower reconfiguration (since reconfiguration time is proportional to bitstream/PRR size). Using more PRRs with smaller RMs can improve reconfiguration time, but can result in more overhead for creating suitably shaped and sized PRRs for diverse RMs. The work in Montone et al. (2010) and Vipin and Fahmy (2013) attempts to automate this process by considering these aspects.

Floorplanning PRRs is also non-trivial as the columnar arrangement of resources in the FPGAs, the need to respect clock domain boundaries, and the need for some overhead to ease placement and routing all complicate the required arrangements. Various approaches to automated floorplanning for PR systems have been proposed. These include mixed integer linear programming formulations (Rabozzi et al. 2016), kernel tessellation (Vipin and Fahmy 2012), and evolving tool placements (Beckhoff et al. 2013).

GoAhead (Beckhoff et al. 2012) automates aspects of bitstream generation including allowing relocation of RMs between PRRs, though it does require user intervention for floorplanning. It automates the insertion of blocker macros to prevent wires from the SR crossing through the PRRs.

CoPR (Vipin and Fahmy 2014) takes a description of valid system configurations and automates the process of determining PRRs, allocating RMs to PRRs, and floorplanning, followed by automation of the build process through scripting of the vendor tools.

BITMAN (Pham et al. 2017) is an open-source tool for generating and manipulating bitstreams. These manipulations can include relocation of bitstreams to different PRRs and modifying individual primitives in the FPGA.

Recently, both AMD/Xilinx and Intel/Altera have added support for nested PRRs. In earlier tools, the designer would need to determine the arrangement of PRRs in advance of design, and these could then not be changed. In the newer flows, PRRs can themselves contain PRRs, allowing the arrangement of lower-level PRRs (*child* partition) to be modified at runtime through reconfiguration of the higher-level PRRs (*parent* partition). For a child partition, the rest of the parent partition is treated like a static partition, even if the parent partition is a PRR. A child RM can be swapped out without affecting the rest of the parent partition. Figure 5 shows a representation of this concept.

The tool flow for hierarchical PR requires another level of floorplanning, netlist, and bitstream generation, including verification of all the possible configurations. RM implementations can be run in parallel once the corresponding static design and floorplan are in place.

AMD/Xilinx also introduced the abstract shell concept, allowing new RMs to be compiled for a PRR without having the full SR available. The abstract shell encapsulates all the required information about the surroundings of the PRR to enable the implementation tools to implement an RM into the PRR. This is shown in Fig. 6.

Despite these improvements, the tool flows still entail some complexities. AMD/Xilinx tools can route wires for the SR through PRRs. As a result, any change in the SR requires all RMs to be re-implemented. Additionally, these reserved wires in the PRRs can make some RMs difficult to route. This is referred to as *bleeding* over the dynamic region (Xilinx 2023). This can be overcome by setting the `CONTAIN_ROUTING` constraint to force routing to stay within the bounding box of the SR. It is also possible to avoid this by placing blockers around the PRR, which prevent wires from entering it (Beckhoff et al. 2012). However, this can cause timing-related issues as wire routes might be longer.

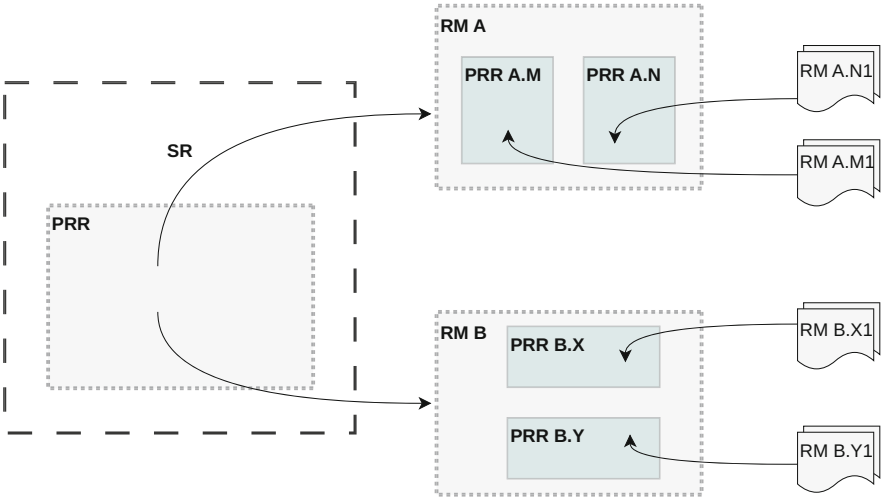


Fig. 5 The nested PR flow with two different nested PRR allocations allowing the location and size of these PRRs to be modified at runtime, thereby enhancing flexibility

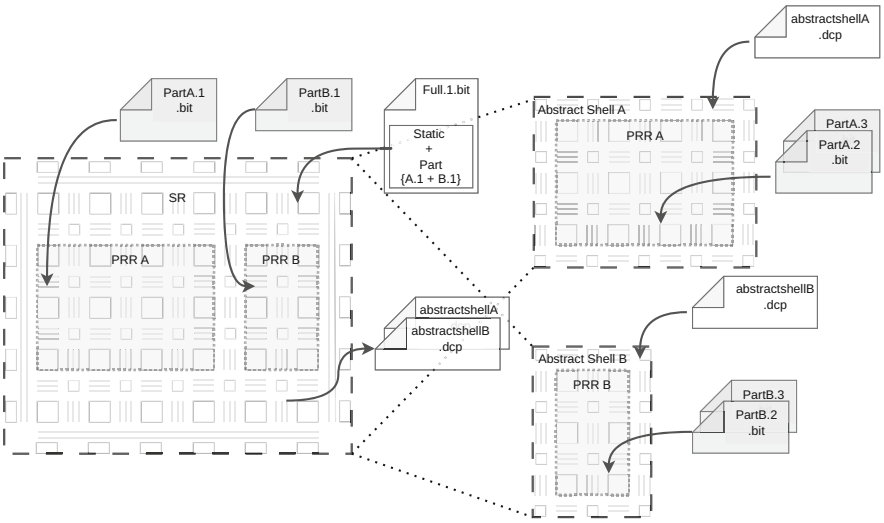


Fig. 6 The abstract shell flow enhances the traditional PR design flow by generating abstract shell checkpoints for each PRR that enable partial bitstreams to be generated without needing the full bitstream

Another key consideration is decoupling signals from/to PRRs and resetting to a known state after reconfiguration. Decoupling helps SR and PRRs to isolate stray signals that might emerge during reconfiguration. This is more complicated for hierarchical PR.

There are a few more considerations specifically targeted at hierarchical PR. Child RMs should only be reconfigured with the corresponding parent RM as there's no verification support built into a configuration controller; the configuration initiator's responsibility is to verify that a child RM corresponds to the current instance of parent RM prior to reconfiguration. Currently, PR and hierarchical PR design tools lack support when it comes to encryption and authentication of bitstreams. Authentication is only supported for full bitstreams, i.e., loading the static region and not for partial bitstreams. Partial bitstreams can be encrypted, but must be with the same key as the static bitstream.

Managing Partial Reconfiguration

Once a design has been compiled and all the full and partial bitstreams generated, the system can use these to switch between hardware configurations. A full bitstream is first loaded, instantiating the SR and potentially an RM in each PRR. (It is possible to create a configuration with an empty RM in each PRR for the initial state.) Partial bitstreams are similar in format to full bitstreams, except they contain only the configuration data associated with their PRRs and are hence smaller. To switch configurations, a partial bitstream must be loaded into the configuration memory. This can be done over any of the suitable configuration interfaces (Table 1). External JTAG is the slowest, averaging 10s of Mbps, while SelectMap (for AMD/Xilinx) and Active Serial (for Intel/Altera) allow for 100s of Mbps depending on the bus width. These interfaces must be controlled by external hardware, such as a distinct processor or externally connected computer through a dedicated programming cable.

For PCIe-hosted FPGAs, AMD/Xilinx provides the MCAP interface, while Intel/Altera provides CvP. These allow a host machine to transfer bitstreams over the PCIe interface. For MPSoC designs, the embedded processor can manage configuration of the programmable FPGA fabric over a controller in the SoC portion of the chip; in the case of AMD/Xilinx, this is called the PCAP.

Table 1 Configuration interfaces and their range of supported bandwidths and resulting approximate configuration times for averagely sized bitstreams. Times marked * are negatively impacted by the overhead of individual word writes. Numbers assume interfaces operate in insecure mode, meaning data is transferred at max bandwidth. Some interfaces support various clock rates

Type	AMD/Xilinx	Intel/Altera	Bandwidth (Gbps)	Reconfig. time (ms)
External	JTAG	JTAG	0.030–0.066	≈10000*
	SelectMap	Active Serial (AS)/FPP	0.66–3.2	≈100*
Internal	ICAP	PR Controller	2–6.4	≈10
PCIe	MCAP	CvP	6.4	≈100*
Processor subsystem	PCAP	FPGA Manager	3.2–6.4	≈30*

AMD/Xilinx offers a primitive called the internal configuration access port (ICAP) which can be instantiated in the SR of a design and which allows bitstream data to be written to the configuration memory. In this way, the device can reconfigure itself. However, the designer must build the required circuitry to move bitstream data, potentially from memory, to this interface.

Configuration controllers expect one data word at a time. Hence, there is a requirement to manage the movement of data into these interfaces, either from host or external processor software, or through a design within the SR. Since the configuration memory is written to through a single interface, multiple PRRs can only be configured sequentially.

Reconfiguration time is a key metric for PR-based designs. Many systems must be able to change modes within a constrained time. Reconfiguration time is determined by the size of the bitstream being configured and the effective throughput of the programming interface. The first depends on the size of the PRR being configured. The second depends on the throughput of the interface and the efficiency of data transfer over the interface (due to the potential control overhead of word-level transfers). ICAP has the highest maximum data throughput of any reconfiguration interface and hence can offer the fastest reconfiguration. However, to achieve this, it is necessary to be able to stream a bitstream to the ICAP at its capable data rate, which requires the design of a custom configuration controller in the SR. It is usually impractical to store even partial bitstreams in on-chip memory due to their size and the limited capacity available. Instead, external DRAM is the most common place to store bitstreams for fast transfer via DMA.

One complication to consider is that partial bitstreams are associated with a specific PRR. Hence, even if the same RM is placed in two different PRRs, that will require two different partial bitstreams to be stored in memory. With multiple RMs mixed into multiple PRRs, the number of different partial bitstreams, and hence the storage capacity required, can increase significantly. There have been efforts to allow bitstreams to be *relocated*. This requires the use of blockers to prevent SR signals to be routed through PRRs or allocation of dedicated SR routing resources through PRRs (Beckhoff et al. 2012). Furthermore, relocation requires low-level knowledge of the FPGA architecture to determine spatial correspondence of different compatible PRRs and modifying the addresses of written configuration words to allow a partial bitstream created for one PRR to be written to another. However, this is unsupported in vendor flows and cannot be applied when bitstream encryption is enabled.

There has been ample work building hardware controllers to manage reconfiguration over the ICAP in AMD/Xilinx devices with the aim of maximizing reconfiguration throughput. The first demonstration of DMA into the ICAP was in Liu et al. (2009), but from on-chip memories, so with very limited storage capacity for bitstreams. This was extended to DMA for off-chip DRAM in Vipin and Fahmy (2012) and then over the PCIe host interface in Vipin and Fahmy (2014a).

Software-managed reconfiguration is the preferred method for FPGA SoCs through the provided PCAP interface on AMD/Xilinx devices for which an API

is provided which can be integrated into user bare metal software running on the processors. However, these deal with passing raw bitstream data to the configuration controller. Reconfiguration using these APIs is a blocking operation, requiring custom designs to overcome this, as was done in Vipin and Fahmy (2014b) where custom interrupts are exploited to allow bitstream DMAs to complete while the processor is busy with other tasks.

Managing PR within an OS running on the host processor requires additional integration. The generally supported approach is a fixed shell into which RMs can be compiled to generate partial bitstreams, which can then be loaded by making calls to an API that manages the transfer of bitstream data. ARTICO3 (Rodríguez et al. 2018) is a framework for the AMD/Xilinx Zynq and ZynqMP MPSoCs that allows custom kernels to be integrated into a software managed platform with automation of access to the memory hierarchy and abstracted PR. Heterogeneous tasks can be exploited to enhance runtime and energy efficiency.

ReConOS (Agne et al. 2014) is a more flexible fully developed OS abstraction for managing PR systems. It tries to unify the management of software and hardware tasks through a unified thread abstraction. Hardware threads are managed through PR and different threads can communicate with each other through various mechanisms. However, the OS kernel must be recompiled to support new hardware threads.

FPGA OS (Vaishnav et al. 2020) decomposes the development of hardware and software for heterogeneous embedded systems while supporting multi-tenancy and abstracted loading of partial bitstreams. However, similar to the above frameworks, this requires all RMs to abide by the uniform shell interface specification. This abstraction works well for pure compute acceleration, but not for some embedded systems where the required interfacing of modules can vary.

ZyPR (Bucknall and Fahmy 2023) extends abstractions further by allowing Linux Device Tree Overlays (DTOs) to be updated dynamically based on the loaded RMs with these being enumerated during the build process. This allows for RMs that access different external I/O to be supported. Additionally, ZyPR presents an abstracted configuration view to the user through its API, with its middleware translating these configuration changes into the required bitstream loading.

Coyote (Korolija et al. 2020) applies various OS abstractions to management of FPGAs, including a process model, scheduling, virtual memory, and I/O virtualization. It also supports partial reconfiguration of FPGA tasks using similar approaches to those described in Section "FPGA Configuration".

Applications of Dynamic Partial Reconfiguration

Having discussed the mechanisms and design approach for dynamic partially reconfigurable systems, how this capability can be exploited in a variety of applications is now discussed in some detail.

Computing Infrastructure and Virtualization

One of the key benefits of FPGAs as a computation platform is that their highly flexible I/O FPGAs can implement a wide range of interfaces that allow them to be integrated into a variety of deployment scenarios. For generalized compute acceleration, this has often been as PCIe accelerators much like GPUs. In such a deployment, a fixed set of FPGA pins is dedicated to implementing the PCIe interface, along with the use of soft or hard PCI interfacing logic in the FPGA. The required accelerator is integrated with this interface logic, and the FPGA can be addressed much like a GPU, as an accelerator for offloading from a host processor. Frameworks like RIFFA (Jacobsen et al. 2015) enabled accelerator designers to automate the process of building the PCIe interface logic and integrating it with their accelerator design, as well as offering software drivers to manage offload. However, each different accelerator would require a full FPGA design build with the integrated PCIe interface logic, and a static reconfiguration of the FPGA, and often, a system reboot, when changing functions.

The DyRACT framework (Vipin and Fahmy 2014a) was the first to use partial reconfiguration to keep the PCIe interface logic active in between reconfigurations of the accelerator logic and to load these bitstreams over PCIe. This concept of an interface shell, containing the required external interfaces to the FPGA and the hardware required to manage reconfiguration in static logic, and the accelerator function in dynamic logic, applied using partial reconfiguration, is now widespread. Microsoft's Catapult project (Caulfield et al. 2016) refers to the static portion as the Blue Bitstream and the user's function as the Green Bitstream. Amazon's AWS F1 (Inc. 2024a) refers to the Shell and Custom Logic. This approach is essential to productive use of FPGAs as accelerators as it means the FPGA's state does not adversely affect the host machine, and that the FPGA can be put to use for diverse application needs as those needs change over time. AMD/Xilinx's Xilinx Runtime Library (XRT) (Inc. 2024b) integrates a host-based runtime with a hardware *platform* composed of the static *Shell* and *User* portions as above. In some of these frameworks, multiple concurrent accelerators are supported, thereby enabling multiple applications to share the FPGA resources and interface access, often referred to as multi-tenancy (Nguyen and Kumar 2020).

In recent years, data center use of FPGAs has increased, and the ability of FPGAs to directly process network packets has become important. FPGA Smart NICs have thus become more commonplace. These are FPGA accelerator cards with PCIe interfaces as above, but with high-speed network interfaces now also serving to interface with the datacenter network. Frameworks like Corundum (Forencich et al. 2020) and AMD OpenNIC (Inc. 2024) offer analogous solutions to RIFFA that extend to the network interface. As of now, these functions are usually integrated in static bitstreams, but the role of reconfiguration to enhance flexibility is expected to see these frameworks extended to support partial reconfiguration soon. Serving as a "bump in the wire" NIC, i.e., passing all packets to a host, while potentially processing a subset of them, requires that any reconfiguration of function does not

disrupt network to host connectivity. Partial reconfiguration is a way to achieve this (Anup Agarwal and Seshan 2023). In Bucknall et al. (2019), the authors demonstrate reconfiguration over a network interface on the AMD/Xilinx Zynq by routing configuration packets directly to the ICAP for high-speed reconfiguration.

Within embedded systems, computing resources are constrained and hardware acceleration becomes an important factor to enable computationally complex applications. However, hardware resources such as the programmable logic(PL) in an FPGA SoC cannot be committed to just individual functions. Instead, in many cases, hardware accelerators should be loaded as needed based on application requirements. Since FPGA SoCs include a software-programmable processor that can even run an operating system, integrating PR becomes important. There are a variety of frameworks that address swapping accelerators, considering the properties of PR and their impact on execution time for complex applications (Steiger et al. 2004). Approaches to support real-time schedules have also been explored (Rossi et al. 2018).

PYNQ (Inc. 2024) from AMD is a framework that provides Python APIs to manage and interact with hardware in the programmable logic (PL) of an FPGA SoC. They refer to the PL configuration as an overlay, and it is loaded via a Python function which loads the corresponding full bitstream into the PL, to instantiate required interfaces and accelerators. Custom drivers are required to be able to access accelerators from Python. PYNQ had some initial experimental approaches for integrating partial reconfiguration (Janßen et al. 2017), which has now been added to the release, necessitating a hierarchical block design to partially reconfigure any region on the FPGA, which are referred to as PYNQ Composable Overlays.

It should be clarified that the use of the term *overlays* in this context simply refers to the different hardware configurations used in the PL of an FPGA SoC. More general compute overlays are usually coarse-grained architectures implemented atop the FPGA (Capalija and Abdelrahman 2013; Jain et al. 2021). These are configured using full reconfiguration then the overlays themselves are configured using a custom interface provided by the overlay. Since the configuration space is so much more constrained, these configuration "bitstreams" are orders of magnitude smaller, do not use the FPGA configuration infrastructure, and hence can be configured orders of magnitude faster than traditional partial reconfiguration (Stitt and Coole 2011).

Design Compilation

A major challenge with FPGA design is the compilation time from synthesis to bitstream generation. This can be hours, or even days, long for complex designs. This is problematic in the development cycle, where even a small change can add many hours to compilation time.

SPADES (Nguyen et al. 2023) demonstrates the effective use of the hard Network on Chip (NoC) in AMD Versal devices, leading to significant gains in compilation

time by separating different parts of the compilation to target independent regions of the FPGA.

HiPR (Xiao et al. 2019) extends this idea using partial reconfiguration. A large design is divided into multiple blocks, where each block is contained in a PRR with minimal overhead. This enables incremental compilation, where only a single PRR needs to be recompiled when a change is made, and parallel compilation, where all modules can be separately compiled in parallel, both of which dramatically reduce design iteration time. They use a packet-switched overlay network for communication between these independent blocks. Placing this network in a PRR eliminates the complex time-consuming routing between different blocks.

In further work Xiao et al. (2022), the authors try to build PR abstractions for high-level synthesis (HLS) to automatically generate static and PR regions. They incorporate the abstract shell design flow for PR functions to be placed and routed in parallel. They show that the potential loss of design optimizations across a large compile is minimal when compared with the productivity benefits.

Adaptive Systems

FPGAs have found widespread use in embedded systems where they can often absorb all the required computing capability to implement complex systems that interact with their environment. This is even more prevalent now with the wide availability of FPGA SoCs that include processor cores on the same fabric, providing software programmability alongside tightly coupled hardware acceleration. Many of these applications in communications, automotive, industrial control, etc. require some level of adaptability to evolving conditions and PR allows these systems to modify their accelerated computing functions as needed at runtime. However, as alluded to in Section "Designing Partially Reconfigurable Systems," the design process can become an obstacle to adoption by domain experts.

Cognitive radio systems have been widely implemented on FPGAs using PR. This application requires a radio system to modify its baseband radio processing algorithms based on adapting requirements. Baseband processing chains are usually implemented in hardware to enable high throughput and low latency. Since the required hardware can be significantly different for different operating modes, multiplexing the various modes can be costly in terms of area. PR allows such radio systems to switch between different modes, e.g., sensing and multiple baseband modes, dynamically at runtime. This allows the radio systems to consume less area and power while still supporting dynamic operation. An example design in Sadek et al. (2017) shows a multi-mode 3G, LTE, and WiFi transceiver using PR to switch modes that reduces power consumption by 66% compared to a multiplexed design while only requiring less than 1.5 ms to switch modes. A similar approach combining PR with module parameterization improves reconfiguration time (Pham et al. 2017).

In vision systems, there can be various forms of data-dependent processing that require different accelerators. Rather than implement all modes and select between

them at runtime, it can be more efficient to dynamically load accelerators. Since the inter-frame interval can be in the order of 10s of milliseconds, it is possible to reconfigure and complete processing within this interval. Nguyen and Hoe (2018) demonstrate this capability by reconfiguring multiple PRRs with different image processing blocks within the time take to process a single frame, achieving 60fps at 720p and 30fps at 1080p resolutions. Nava et al. (2011) demonstrate a similar approach in the vision system of a robot, where, depending on the number of colors detected, different processing modes are enabled using PR. Figure 7 shows an example setup of a partially reconfigurable video processing system.

FPGA PR has also been exploited in networking applications. A Ternary Content-Addressable Memory (TCAM) architecture using PR was presented in Reviriego et al. (2019). TCAMs are used for complex pattern matching and lookups in software-defined networking applications but must be emulated on the FPGA architecture components. The authors showed that PR could be used to reduce the area consumption of TCAM designs by over 35% while still supporting update rates of several hundred or thousand rules per second. The authors in Li et al. (2018) use PR to enable network function virtualization (NFV) using a framework called DHL that they propose. It allows virtual network functions to run in software while offloading computationally intensive portions of code to an FPGA, using PR to support multiple functions.

Adaptive systems must be agile as they are deployed in unknown environments sometimes requiring a change in processing hardware. The Observe-Decide-Act

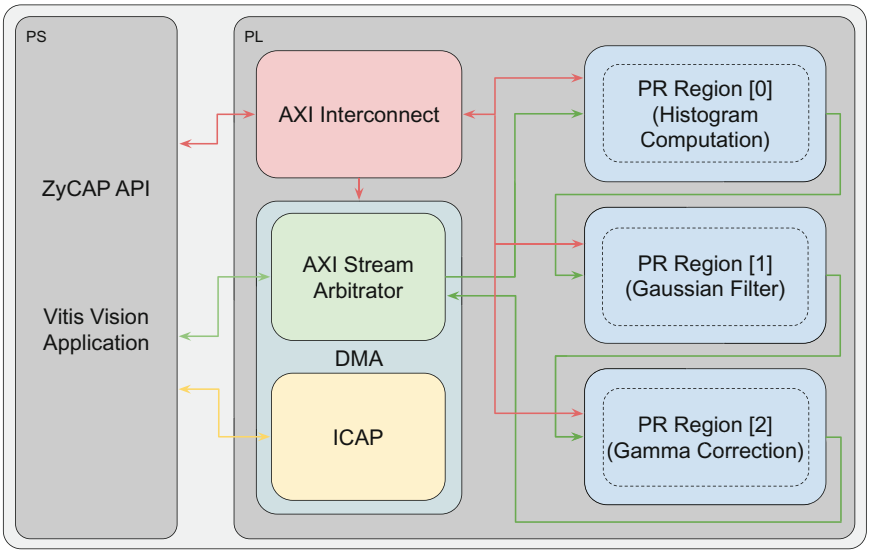


Fig. 7 A vision application on an AMD/Xilinx Zynq UltraScale+ MPSoc device, showing the PS running software that adapts the types of vision filters executed in the PL using PR (Bucknall 2022)

loop typical in adaptive systems can be mapped well to FPGA SoCs where context switches can be applied through PR (Fahmy 2018). PR can also support various robustness and failure recovery modes in such systems (Paulsson et al. 2006).

Machine Learning

Significant work has been conducted on optimizing deep neural network accelerator hardware. These optimizations often require mixed or custom numerical precision support, and FPGAs are ideally suited to such hardware designs. FPGAs have been used to implement accelerators for models with precision ranging from 32-bit floating point, through various fixed point representations, down to binary neural networks, being able to trade off area against a tolerable accuracy loss.

In Hussain et al. (2015), the authors build a multi-classifier system that can switch between support vector machine (SVM) and K nearest neighbor (KNN) classifiers dynamically at runtime and demonstrate that this is up to 8× faster than traditional full bitstream reconfiguration.

In Irmak et al. (2021), the authors demonstrate how PR can be used to dynamically switch convolutional neural network (CNN) architectures for multiple applications. They show that having optimized CNNs for different applications and swapping them using PR is preferable in terms of accuracy to training a hybrid CNN while still offering low latency.

In Venieris and Bouganis (2017), the authors build a CNN design framework that allows multiple blocks of a CNN to be allocated to the same hardware accelerator through weight only reconfiguration while amortizing the more costly partial reconfiguration for other blocks through using larger batch sizes to achieve low latency.

CascadeCNN (Kouris et al. 2018) switches between accelerator hardware with different numerical precision and hence accuracy. If a sample is misclassified, it is recomputed with a higher precision model which is slower; otherwise, the faster inference is accepted. Partial reconfiguration is suggested as a way to switch between these models.

Reliability and Harsh Environments

Designing electronic systems for harsh environments is challenging. Ionizing radiation affects the properties of transistors, and high-energy particles can damage the materials themselves, all resulting in failures. These long-term effects are alongside immediate *single event effects* (SEEs) which can occur due to high-energy particles colliding with a device. FPGAs suffer in the same way as other devices, but beyond effects in the implemented circuit, the configuration memory is also susceptible to corruption. This depends on how the configuration memory is implemented (Wirthlin 2015). Antifuse and Flash configuration memories are generally robust to SEEs, though Flash is more susceptible to longer-term ionizing

doses. SRAM FPGAs, however, which are more prevalent, can suffer single event upsets (SEUs) in their configuration memory, a change in a bit's state. This is highly problematic since that can modify the configured datapath and even lead to an incorrectly configured FPGA.

Space is a commonly considered harsh environment. FPGAs are popular in space applications due to increasing processing requirements and the need for custom processing architectures (Osterloh et al. 2009). Radiation in space is a result of cosmic rays or high-energy particles which can strike devices. FPGAs are also widely used in high-energy physics experiments where they are used to process detector data to extract important features for further analysis. In such experiments, the radiation environment is driven by the types of experiments being performed, which typically involved strong radiation fields and high-energy particles.

There are a variety of approaches for tackling reliability in such environments, including spatial redundancy, such as triple modular redundancy (TMR). However, a solution unique to FPGAs is configuration scrubbing. This is where the configuration memory is repeatedly rewritten to combat potential SEUs. It is possible to read the contents of this memory and rewrite it periodically, or else to continuously write the known good configuration (Stoddard et al. 2016). This approach remains imperfect since it is periodic and takes time, so it is typically combined with spatial approaches (Bolchini et al. 2007; Ichinomiya et al. 2010). An automated framework for isolated partially reconfigurable modules with TMR was presented in Pham et al. (2018). In Iturbe et al. (2011), the authors present the R3TOS framework that combines a runtime scheduler with reliability heuristics to allocate hardware tasks in a way that minimizes the effects of SEUs and ionizing radiation.

FPGAs can also offer enhanced robustness to failures. In Dörr et al. (2019) the authors demonstrate the use of PR to manage a fallback processor to support fail-operation in a complex system. In Oszwald et al. (2018), they demonstrate that this capability can meet safety requirements for automotive applications. In Shreejith et al. (2013), PR is used to recover from faults while a redundant unit manages data processing. This is done at the network level to minimize time overhead.

FPGAs can also serve as a fallback mechanism for any arbitrary faults in the system, potentially saving costs associated with adding redundancy to every sub-unit. For instance, consider a system with multiple control units that can be spawned on-the-fly within an FPGA. Instead of implementing Triple Module Redundancy (TMR), an FPGA as a fallback option to emulate the faulty unit can be employed.

Research Directions

Recent focus on open-source tool flows for FPGAs has highlighted the obstacles presented by proprietary bitstream formats. Efforts to address this are gaining some attention. The potential to create bitstream formats that offer more flexibility such as relocatability and runtime modification can further simplify the design and management of PR systems.

While the vendor design processes have improved, mapping these to higher-level system design paradigms useful to embedded and adaptive systems designers remains a challenge. Most frameworks discussed still require the application user to know which bitstreams correspond to a particular RM and PRR. An abstracted model that addresses system designers without hardware experience would increase the likelihood that they adopt PR in their designs.

The security considerations of partial reconfiguration in the context of multi-tenancy systems are also under exploration. Considering multiple users sharing a single FPGA, it is possible for side channels to enable information leakage or other attacks which must be mitigated (Ahmed et al. 2022).

Conclusions

Partial reconfiguration of FPGAs has been researched for two decades, yet still remains a niche feature in practical use. While there remain challenges, the enhanced vendor design flows and the increasing requirement for FPGAs to address general-purpose compute acceleration in challenging scenarios mean that PR is gaining increased attention. Present development shell concepts now provided by vendors use PR in the background, demonstrating its robustness and applicability. Recent developments like the abstract shell design flow make the idea of evolving functionality of PR systems after initial deployment feasible. Hierarchical PR also reduces the previous limitations of a single static determination of PRRs on the device. Using these developments, system designers can now apply PR in more contexts. The research community is expected to exploit such developments to further simplify and streamline PR system design leading to potential wider adoption.

References

Agne A, Happe M, Keller A, Lübbers E, Plattner B, Platzner M, Plessl C (2014) Reconos: an operating system approach for reconfigurable computing. IEEE Micro 34(1):60–71

Ahmed MK, Mandebi J, Saha SK, Bobda C (2022) Multi-tenant cloud FPGA: a survey on security. arXiv preprint arXiv:2209.11158

Anup Agarwal DK, Seshan S (2023) StaRRNIC: Enabling Runtime Reconfigurable FPGA NICs. http://reports-archive.adm.cs.cmu.edu/anon/2023/CMU-CS-23-100.pdf

Beckhoff C, Koch D, Torresen J (2012) GoAhead: a partial reconfiguration framework. In: Proceedings of the IEEE international symposium on field-programmable custom computing machines (FCCM), pp 37–44

Beckhoff C, Koch D, Torreson J (2013) Automatic floorplanning and interface synthesis of island style reconfigurable systems with GoAhead. In: International conference on architecture of computing systems, pp 303–316

Beckhoff C, Koch D, Torresen J (2014) Portable module relocation and bitstream compression for Xilinx FPGAs. In: 2014 24th international conference on field programmable logic and applications (FPL), pp 1–8

Benz F, Seffrin A, Huss SA (2012) Bil: a tool-chain for bitstream reverse-engineering. In: International conference on field programmable logic and applications (FPL), pp 735–738

Bolchini C, Miele A, Santambrogio MD (2007) TMR and partial dynamic reconfiguration to mitigate SEU faults in FPGAs. In: IEEE international symposium on defect and fault-tolerance in VLSI Systems (DFT), pp 87–95

Boutros A, Betz V (2023) Field-programmable gate array architecture. In: Chattopadhyay A (ed) Handbook of computer architecture. Springer, Singapore

Bucknall AR, Shreejith S, Fahmy SA (2019) Network enabled partial reconfiguration for distributed FPGA edge acceleration. In: 2019 international conference on field-programmable technology (ICFPT), pp 259–262

Bucknall AR, Fahmy SA (2023) ZyPR: end-to-end build tool and runtime manager for partial reconfiguration of FPGA SoCs at the edge. ACM Trans Reconfigurable Technol Syst (TRETS) 16(3):34–13433

Bucknall AR (2022) Build framework and runtime abstraction for partial reconfiguration on FPGA SoCs. Phd thesis, University of Warwick

Capalija D, Abdelrahman TS (2013) A high-performance overlay architecture for pipelined execution of data flow graphs. In: Proceedings of the international conference on field programmable logic and applications (FPL)

Caulfield AM, Chung ES, Putnam A, Angepat H, Fowers J, Haselman M, Heil S, Humphrey M, Kaur P, Kim J-Y et al (2016) A cloud-scale acceleration architecture. In: IEEE/ACM international symposium on microarchitecture (MICRO)

Compton K, Hauck S (2002) Reconfigurable computing: a survey of systems and software. ACM Comput Surv 34(2):171–210

Dörr T, Sandmann T, Schade F, Bapp FK, Becker J (2019) Leveraging the partial reconfiguration capability of FPGAs for processor-based fail-operational systems. In: International symposium on applied reconfigurable computing, pp 96–111

Duncan A, Rahman F, Lukefahr A, Farahmandi F, Tehranipoor M (2019) FPGA bitstream security: a day in the life. In: IEEE international test conference (ITC), pp 1–10

Fahmy SA (2018) Design abstraction for autonomous adaptive hardware systems on FPGAs. In: NASA/ESA conference on adaptive hardware and systems (AHS), pp 142–147

Forencich A, Snoeren AC, Porter G, Papen G (2020) Corundum: an open-source 100-GBPS NIC. In: IEEE international symposium on field-programmable custom computing machines (FCCM), pp 38–46

Hussain HM, Benkrid K, Seker H (2015) Dynamic partial reconfiguration implementation of the SVM/KNN multi-classifier on FPGA for bioinformatics application. In: Proceedings of the annual international conference of the IEEE engineering in medicine and biology society (EMBC), pp 7667–7670

Ichinomiya Y, Tanoue S, Amagasaki M, Iida M, Kuga M, Sueyoshi T (2010) Improving the robustness of a softcore processor against SEUs by using TMR and partial reconfiguration. In: IEEE international symposium on field-programmable custom computing machines (FCCM), pp 47–54

Inc. A (2024a) AWS-FPGA. https://github.com/aws/aws-fpga

Inc. X (2024b) XRT. https://github.com/Xilinx/XRT

Inc. X (2024) Open-NIC. https://github.com/Xilinx/open-nic

Inc. X (2024) PYNQ. https://github.com/Xilinx/PYNQ

Irmak H, Ziener D, Alachiotis N (2021) Increasing flexibility of FPGA-based CNN accelerators with dynamic partial reconfiguration. In: Proceedings of the international conference on field-programmable logic and applications (FPL), pp 306–311

Iturbe X, Benkrid K, Arslan T, Hong C, Erdogan AT, Martinez I (2011) Enabling FPGAs for future deep space exploration missions: improving fault-tolerance and computation density with R3TOS. In: NASA/ESA conference on adaptive hardware and systems (AHS), pp 104–112

Intel (2021) Using the Design Security Features in Intel®FPGAs. https://www.intel.com/content/www/us/en/docs/programmable/683269/current/using-the-design-security-features-in-fpgas.html

Jacobsen M, Richmond D, Hogains M, Kastner R (2015) RIFFA 2.1: a reusable integration framework for FPGA accelerators. ACM Trans Reconfigurable Technol Syst (TRETS) 8(4):1–23

Jain AK, Maskell DL, Fahmy SA (2021) Coarse grained FPGA overlay for rapid just-in-time accelerator compilation. IEEE Trans Parallel Distrib Syst 33(6):1478–1490

Janßen B, Zimprich P, Hübner M (2017) A dynamic partial reconfigurable overlay concept for PYNQ. In: Proceedings of the international conference on field programmable logic and applications (FPL)

Karen Horovitz RK (2024) Intel FPGA Secure Device Manager. https://apps.dtic.mil/sti/pdfs/AD1052301.pdf

Koch D (2012) Partial reconfiguration on FPGAs: architectures, tools and applications, vol 153. Springer, New York

Korolija D, Roscoe T, Alonso G (2020) Do OS abstractions make sense on FPGAs? In: 14th USENIX symposium on operating systems design and implementation (OSDI 20). USENIX Association, pp 991–1010

Kouris A, Venieris SI, Bouganis C-S (2018) CascadeCNN: pushing the performance limits of quantisation in convolutional neural networks. In: 2018 28th international conference on field programmable logic and applications (FPL), pp 155–1557

Kuon I, Tessier R, Rose J (2008) FPGA architecture: survey and challenges. Found Trends Electron Des Autom 2:135–253

Li X, Wang X, Liu F, Xu H (2018) DHL: enabling flexible software network functions with FPGA acceleration. In: IEEE international conference on distributed computing systems (ICDCS)

Liu M, Kuehn W, Lu Z, Jantsch A (2009) Run-time partial reconfiguration speed investigation and architectural design space exploration. In: International conference on field programmable logic and applications (FPL), pp 498–502

Montone A, Santambrogio MD, Sciuto D, Memik SO (2010) Placement and floorplanning in dynamically reconfigurable FPGAs. ACM Trans Reconfigurable Technol Syst (TRETS) 3(4):1–34

Nava F, Sciuto D, Santambrogio MD, Herbrechtsmeier S, Porrmann M, Witkowski U, Rueckert U (2011) Applying dynamic reconfiguration in the mobile robotics domain: a case study on computer vision algorithms. ACM Trans Reconfigurable Technol Syst (TRETS) 4(3), 29:1–29:22

Nguyen TD, Kumar A (2020) Maximizing the serviceability of partially reconfigurable FPGA systems in multi-tenant environment. In: Proceedings of the ACM/SIGDA international symposium on field-programmable gate arrays, pp 29–39

Nguyen T, Blair Z, Neuendorffer S, Wawrzynek J (2023) SPADES: A productive design flow for Versal programmable logic. In: 2023 33rd international conference on field-programmable logic and applications (FPL), pp 65–71

Nguyen M, Hoe JC (2018) Time-shared execution of realtime computer vision pipelines by dynamic partial reconfiguration. In: International conference on field programmable logic and applications (FPL), pp 230–2304

Osterloh B, Michalik H, Habinc SA, Fiethe B (2009) Dynamic partial reconfiguration in space applications. In: NASA/ESA conference on adaptive hardware and systems, pp 336–343

Oszwald F, Becker J, Obergfell P, Traub M (2018) Dynamic reconfiguration for real-time automotive embedded systems in fail-operational context. In: IEEE international parallel and distributed processing symposium workshops, pp 206–209

Paulsson K, Hubner M, Becker J (2006) Strategies to on-line failure recovery in self-adaptive systems based on dynamic and partial reconfiguration. In: First NASA/ESA conference on adaptive hardware and systems (AHS'06), pp 288–291

Pham K, Horta E, Koch D, Vaishnav A, Kuhn T (2018) IPRDF: an isolated partial reconfiguration design flow for Xilinx FPGAs. In: International symposium on embedded multicore/many-core systems-on-chip (MCSoC), pp 36–43

Pham TH, Fahmy SA, McLoughlin IV (2017) An end-to-end multi-standard OFDM transceiver architecture using FPGA partial reconfiguration. IEEE Access 5:21002–21015

Pham KD, Horta E, Koch D (2017) BITMAN: a tool and API for FPGA bitstream manipulations. In: Design, automation and test in Europe conference and exhibition (DATE), pp 894–897

Proulx A, Chouinard J-Y, Fortier P, Miled A (2023) A survey on FPGA cybersecurity design strategies. ACM Trans Reconfigurable Technol Syst 16(2):1–33

Rabozzi M, Durelli GC, Miele A, Lillis J, Santambrogio MD (2016) Floorplanning automation for partial-reconfigurable FPGAs via feasible placements generation. IEEE Trans Very Large Scale Integr (VLSI) Syst 25(1):151–164

Reviriego P, Ullah A, Pontarelli S (2019) PR-TCAM: efficient TCAM emulation on Xilinx FPGAs using partial reconfiguration. IEEE Trans Very Large Scale Integr (VLSI) Syst 27(8):1952–1956

Rodríguez A, Valverde J, Portilla J, Otero A, Riesgo T, Torre E (2018) FPGA-based high-performance embedded systems for adaptive edge computing in cyber-physical systems: the ARTICo3 framework. Sensors 18(6):1877

Rossi E, Damschen M, Bauer L, Buttazzo G, Henkel J (2018) Preemption of the partial reconfiguration process to enable real-time computing with FPGAs. ACM Trans Reconfigurable Technol Syst (TRETS) 11(2):1–24

Sadek A, Mostafa H, Nassar A Ismail Y (2017) Towards the implementation of multi-band multi-standard software-defined radio using dynamic partial reconfiguration. Int J Commun Syst 30(17):3342

Shreejith S, Vipin K, Fahmy SA, Lukasiewycz M (2013) An approach for redundancy in FlexRay networks using FPGA partial reconfiguration. In: Design, automation and test in Europe conference and exhibition (DATE), pp 721–724

Soni RK, Steiner N, French M (2013) Open-source bitstream generation. In: IEEE international symposium on field-programmable custom computing machines (FCCM), pp 105–112

Steiger C, Walder H, Platzner M (2004) Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks. IEEE Trans Comput 53(11):1393–1407

Stitt G, Coole J (2011) Intermediate fabrics: virtual architectures for near-instant FPGA compilation. IEEE Embed Syst Lett 3(3):81–84

Stoddard A, Gruwell A, Zabriskie P, Wirthlin MJ (2016) A hybrid approach to FPGA configuration scrubbing. IEEE Trans Nucl Sci 64(1):497–503

Vaishnav A, Pham KD, Koch D (2018) A survey on FPGA virtualization. In: International conference on field programmable logic and applications (FPL), pp 131–138

Vaishnav A, Pham K, Powell J, Koch D (2020) Fos: a modular FPGA operating system for dynamic workloads. ACM Trans Reconfigurable Technol Syst 20:1–20:28

Vipin K, Fahmy SA (2012) Architecture-aware reconfiguration-centric floorplanning for partial reconfiguration. In: Reconfigurable computing: architectures, tools and applications: international symposium on applied reconfigurable computing, pp 13–25

Vipin K, Fahmy SA (2012) A high speed open source controller for FPGA partial reconfiguration. In: International conference on field-programmable technology (FPT), pp 61–66

Vipin K, Fahmy SA (2013) Automated partitioning for partial reconfiguration design of adaptive systems. In: IEEE international symposium on parallel and distributed processing workshops, pp 172–181

Vipin K, Fahmy SA (2014) Automated partial reconfiguration design for adaptive systems with CoPR for Zynq. In: Proceedings of the international symposium on field-programmable custom computing machines (FCCM), pp 202–205

Vipin K, Fahmy SA (2014a) DyRACT: a partial reconfiguration enabled accelerator and test platform. In: International conference on field programmable logic and applications (FPL)

Vipin K, Fahmy SA (2014b) ZyCAP: efficient partial reconfiguration management on the Xilinx Zynq. IEEE Embed Syst Lett 6(3):41–44

Vipin K, Fahmy SA (2018) FPGA dynamic and partial reconfiguration: a survey of architectures, methods, and applications. ACM Comput Surv (CSUR) 51(4), 72:1–72:39

Venieris SI, Bouganis C-S (2017) Latency-driven design for FPGA-based convolutional neural networks. In: Proceedings of the international conference on field programmable logic and applications (FPL)

Vliegen J, Mentens N, Verbauwhede I (2013) A single-chip solution for the secure remote configuration of FPGAs using bitstream compression. In: International conference on reconfigurable computing and FPGAs (ReConFig), pp 1–6

Wirthlin M (2015) High-reliability FPGA-based systems: space, high-energy physics, and beyond. Proc IEEE 103(3):379–389

Xiao Y, Park D, Butt A, Giesen H, Han Z, Ding R, Magnezi N, Rubin R, DeHon A (2019) Reducing FPGA compile time with separate compilation for FPGA building blocks. In: 2019 international conference on field-programmable technology (ICFPT), pp 153–161. https://doi.org/10.1109/ICFPT47387.2019.00026

Xiao Y, Hota A, Park D, DeHon A (2022) HiPR: high-level partial reconfiguration for fast incremental FPGA compilation. In: 2022 32nd international conference on field-programmable logic and applications (FPL), pp 70–78

Xilinx (2023) UltraScale Architecture Configuration User Guide. https://docs.amd.com/v/u/en-US/ug570-ultrascale-configuration

Xilinx (2023) Abstract Shell for Dynamic Function eXchange. https://docs.xilinx.com/r/en-US/ug909-vivado-partial-reconfiguration/Abstract-Shell-for-Dynamic-Function-eXchange

Xilinx (2024) Using Encryption and Authentication to Secure an UltraScale/UltraScale+ FPGA Bitstream Application Note (XAPP1267). https://docs.xilinx.com/r/en-US/xapp1267-encryp-efuse-program/Using-Encryption-and-Authentication-to-Secure-an-UltraScale/UltraScale-FPGA-Bitstream-Application-Note

GPU Architecture

16

Hyeran Jeon

Contents

Introduction 532
Graphics Pipeline 533
GPU for General-Purpose Computing 536
  Execution Model 536
  Programming Interface 538
  Hardware Architecture 540
  Memories 545
Recent Research on GPU Architecture 550
  Performance 550
  Energy Efficiency 552
  Reliability 555
Conclusion 557
References 557

Abstract

The graphics processing unit (GPU) became an undoubtedly important computing engine for high-performance computing. With massive parallelism and easy programmability, GPU has been quickly adopted by various emerging computing domains including gaming, artificial intelligence, security, virtual reality, and so on. With its huge success in the market, GPU execution and its architecture became one of the essential topics in parallel computing today. The goal of this chapter is to provide readers with a basic understanding of GPU architecture and its programming model. This chapter explores the historical background of current GPU architecture, basics of various programming interfaces, core

H. Jeon (✉)
University of California Merced, Merced, CA, USA
e-mail: hjeon7@ucmerced.edu

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_66

architecture components such as shader pipeline, schedulers and memories that support SIMT execution, various types of GPU device memories and their performance characteristics, and some examples of optimal data mapping to memories. Several recent studies are also discussed that helped advance the GPU architecture from the perspectives of performance, energy efficiency, and reliability.

Keywords

GPU · SIMT · Parallel computing platform

Introduction

Graphics processing units (GPUs) are one of the most widely used accelerators today. As of 2021, 7 out of top 10 world-class super computers are powered by GPUs (Top500 2021). After gaining popularity in high-performance computing domain since mid-2000s, GPU has been conquering emerging domains such as deep learning, security, virtual reality, and so on by proving superior performance than other general-purpose computing platforms, easier programmability than specialized accelerators, and higher affordability than server systems. It is now impossible to understand the performance of virtually most of the computing systems, from servers to mobile devices, without a knowledge about GPU. This chapter aims at providing a thorough description of the full stack of GPU computing, from execution model and programming interfaces to hardware architecture details that includes organization of compute cores and memory subsystems. The readers will be able to grasp unique characteristics of GPU computing and the architecture. However, the limited space was not sufficient to cover all the latest designs of this quickly evolving architecture. Thus, this chapter focuses more on describing the fundamental architecture components and their design details that have been maintained across all generations of GPU architectures, such as SIMT execution, batched processing (in warp or wavefront), diverging memory types and their characteristics, etc. A few recent studies are also introduced that motivated architectural advances. The authors hope that this chapter can be used for developing a basic understanding and finding ways to navigate advanced features of GPUs.

The chapter is structured as follows. In section "Graphics Pipeline," the graphics pipeline is overviewed by exploring the core functions of graphics processing and the architecture of the traditional GPU that only supports graphics applications. This section will provide historical background of the baseline architecture of GPUs today. The readers can understand the limitations of traditional GPUs and how the mitigating efforts led to a brand-new architecture that made GPUs become one of the most important high-performance computing engines.

In section "GPU for General-Purpose Computing," the full stack of general-purpose GPU (GPGPU) computing is introduced, from execution model to

microarchitecture components. The section begins with describing two-level parallelism and show example GPU programs in different programming interfaces. Section "Hardware Architecture" introduces computing components in GPU architecture such as overall processor organization, shader pipeline, banked register file, warp scheduler, and SIMT stack. Some of the unique architectural characteristics and their limitations described in this section will be useful for understanding the research trend discussed in section "Recent Research on GPU Architecture." Section "Memories" explains the types of GPU memories and the characteristics of each of them. It shows how to utilize different memories according to the access patterns of individual data by using an example code in section "Optimization Use Case: Access-Aware Variable Mapping to Memory."

Section "Recent Research on GPU Architecture" discusses research trends to improve performance, energy efficiency, and reliability of GPU architecture. Due to the important role of GPUs to many computing fields, GPU architecture is one of the most actively researched domains in last decade. Due to the limited space, only limited number of important studies are included in this chapter. The authors hope the introduced assorted studies make readers to gain good insights and help the readers to navigate related work.

Graphics Pipeline

As GPUs were originally designed for handling graphics processing, traditional GPUs were equipped with a few specialized cores dedicatedly designed for the graphics pipeline. The common pipeline steps of graphics processing are vertex, geometry, pixel, and rendering, as illustrated in Fig. 1. Vertex step recognizes the end points of edges of an object in the virtual space to two-dimensional screen. Geometry step identifies the curves and lines that connect any two vertexes. Pixel step fills each unit space on the surface recognized by the former two steps with color values. Rendering step smoothen the color and the shape of the surfaces to make the objects look more realistic. The rendering output is compressed to be shown on the screen through framebuffer. For better understanding, suppose that one has two triangles to show on the screen. The vertex step identifies the three edge points of each triangle. Then, the geometry step draws lines between any two points that belong to a triangle. The pixel step fills the pixels within each triangle boundary with specified color values, one red and another blue with gradients in the example of Fig. 1. After the rendering step smoothens out the boundaries of pixels, the screen projections of the two triangles are sent to the framebuffer to be shown on the display device (e.g., a monitor).

To fulfill the aforementioned graphics processing steps, GPUs used to employ dedicated processing cores, namely, vertex core, geometry core, and pixel core. These dedicated cores were cost effective for the traditional GPUs that were used only for graphics processing because unnecessary logics do not need to be

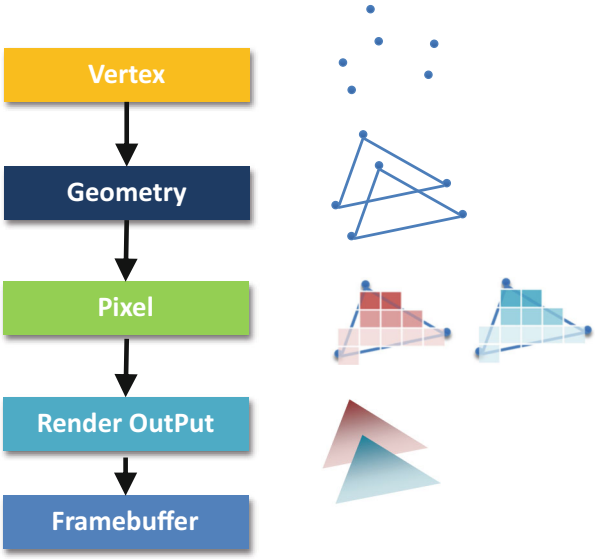


Fig. 1 Common pipeline of graphics processing

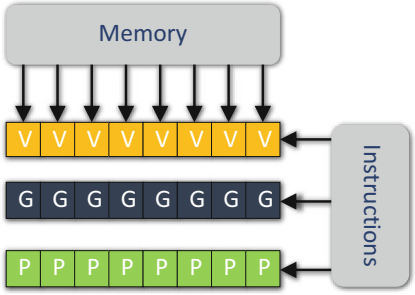


Fig. 2 Simplified architecture of traditional GPU architecture

implemented. Figure 2 shows a simplified GPU architecture where each core is marked with the first character of its function (e.g., V core is for vertex processing). Each core was connected to instruction and data memories and executes the specified functions on one (or multiple) data point(s). The number of cores of individual functions are statically determined at the fabrication time and hence the degree of parallelism is limited to the provided core counts. This static design is simple, but there was a shortcoming of performance imbalance depending on the input images.

Figure 3 shows two example input images that have different characteristics. The left-hand side image includes an object that has a very complex structure. On the other hand, the right-hand side image shows a relatively simple landscape view but

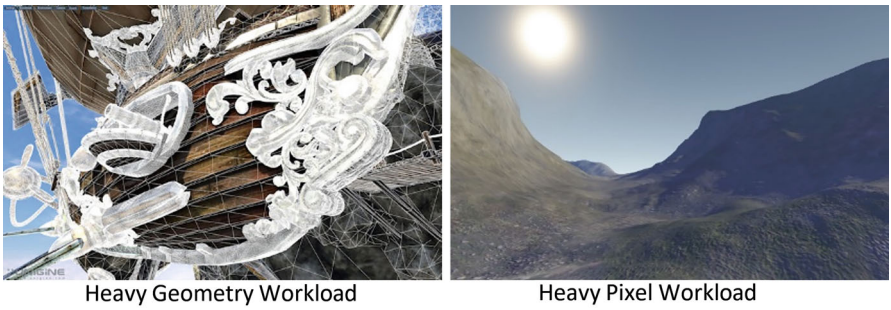


Fig. 3 Different image inputs that cause underutilization in GPU architecture

painted with complicated color variants. Though the required steps of processing of these two images are identical, it is obvious that each of them needs different functional intensities. The left-hand side image needs more operations of geometry functions while the right-hand side image has higher demand on pixel functions. If these two images are processed on the same GPU that has fixed number of geometry and pixel cores, the types of cores that the input image has higher intensity will become the performance bottleneck. As the number of cores per function is not dynamically reconfigurable, the throughput of this traditional GPUs was highly dependent on the types of input images.

To resolve the aforementioned performance issue, one solution was to combine the dedicated functional cores into a uniform core, namely the *unified shader core*. The unified shader core can handle basic arithmetic and logic operations similar to arithmetic logic units (ALU). Therefore, the graphics operations can be executed by using combinations of instructions. This effectively resolves the performance issue caused by limited core resources because each graphics function can use all cores for its computation. This unified shader core also shed lights on GPUs for general-purpose computing, which is called GPGPU. As unified shader cores can not only run graphics operations but also handle any arithmetic operations, they can also be used for running general-purpose applications such as sorting, matrix/vector operations, etc. As GPUs inherently have massive parallel processing powers with tens of compute cores to handle abundant pixel processing, a new computing wave raised to leverage GPUs with unified shader core as a new high-performance computing platform for fulfilling the performance needs of big data workloads that were challenging to support with handful of CPU cores. With the releases of software frameworks that provide C-like programmability for GPUs, GPGPU became one of the most promising high-performance computing platforms since mid-2000s. The GPGPU programming model and architectures will be explored in the following section. As virtually most of the modern GPUs have general-purpose computing capabilities, GPGPU and GPU will be used interchangeably in the following sections.

GPU for General-Purpose Computing

Execution Model

With unified shader cores, GPU is adopted for non-graphics domains especially for high-performance computing. GPUs have demonstrated superior performance for data-intensive workloads that need massive data-level parallelism. Unlike CPUs that are designed for handling complex control flows with high instruction-level parallelism, GPUs are mainly used for the applications that need to process abundant data. So-called big data workloads, non-graphics GPU applications have similarities with graphics applications in the way that large volumes of independent data (pixels in graphics) need to be processed with the same algorithms (the core functions of the graphics pipeline). For example, deep learning is proven to be well suited to GPU computing. In deep learning, tens to hundreds of data in each layer of a neural network are processed with neurons that run the same algorithm (e.g., convolution function). Data within a neural network layer are independent and hence neurons do not need to consider inter-neuron data dependency. As far as dependencies across layers are enforced, all data in each layer can be computed with identical functions independently. Such GPU's execution model is similar to single-instruction multiple-data (SIMD) in Flynn's taxonomy because multiple independent data are processed with the same algorithm. Unlike vector processors that implement SIMD with single instruction fetch for a vector of data, GPUs use multiple threads, each fetches the same instruction to process one datum. Therefore, GPU computing is more precisely defined as single-instruction multiple threads (SIMT).

Even with unified shader core with SIMT processing, there was another performance bottleneck that is data access latency. Figure 4a shows a simplified SIMT execution where a group of SIMT lanes (white lines in the computation box) encounter a memory stall and resume the execution once the data arrive from the memory. As all threads execute the same instruction each cycle, all threads within the SIMT unit encounter memory stalls at the same time. Then, the threads stop execution until when the data become ready. During that time, all shader cores become idle. Because memory access latency is typically hundreds of cycles, the performance overhead is significant. Therefore, another evolution was needed in the thread scheduling. Figure 4b shows the group-level thread scheduling, where a GPU runs a program with groups of threads and schedules each group in a certain order. The example figure shows a round-robin scheduling where thread groups 1 to 3 take turns to run computations with time multiplexing. The thread group context switching happens when each group encounters memory stalls. While one thread group (e.g., group 1 in the figure) is waiting for their data, another ready group (e.g., group 2) runs non-memory operations. If enough number of thread groups are supported, the memory stall overhead can be completely hidden. This thread groups are called *warp* in NVIDIA GPUs and *wavefront* in AMD GPUs. Warp scheduling is handled by hardware schedulers that are called warp scheduler or wavefront schedulers. The details of warp scheduling will be discussed in section "Hardware Architecture."

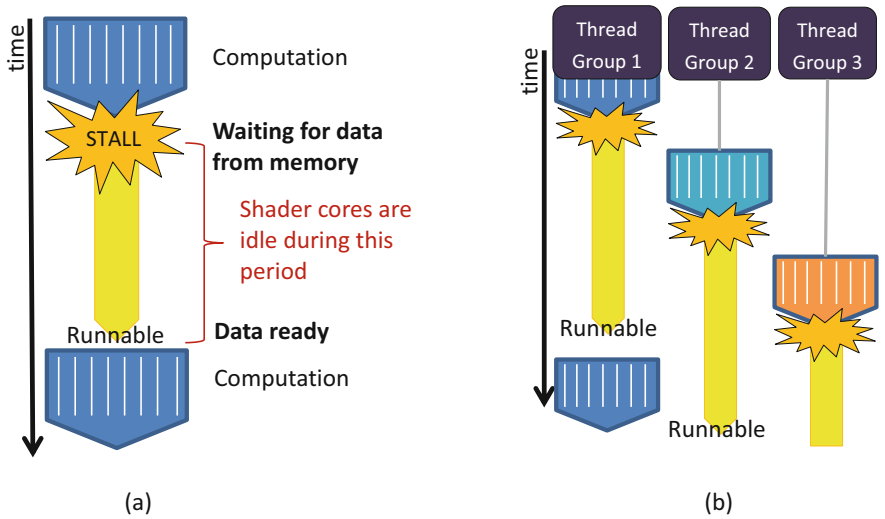


Fig. 4 GPU execution (**a**) without thread groups and (**b**) with batched processing in thread groups

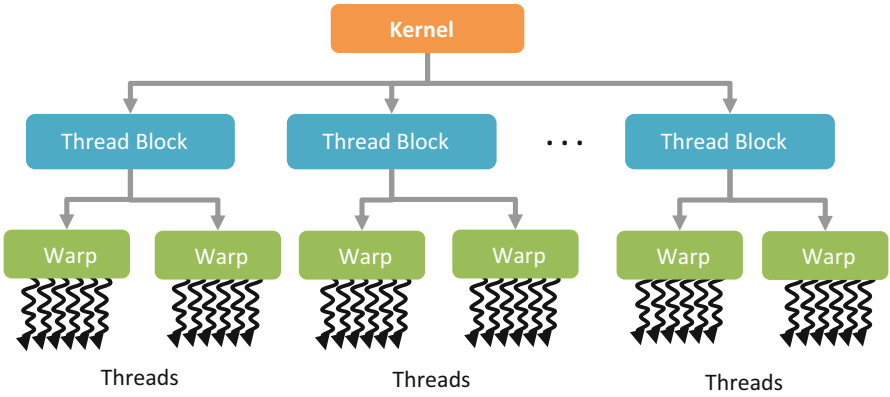


Fig. 5 Hierarchical GPU execution model

GPU has a hierarchical execution model, as illustrated in Fig. 5. The entry function of GPU program is called *kernel*. The kernel is executed in multiple *thread blocks* or *work groups*. Each thread block consists of multiple *warps* or *wavefronts* where the warp is a group of *threads* or *work items* that run the same instruction concurrently. Thread block is the unit of logical data sharing, which means that the threads within the same thread block can share data through on-chip memory. On the other hand, thread blocks are executed independently, which means that threads in different thread blocks do not have an interface to communicate with each other through on-chip memory. This hierarchical execution model maps well with the

structures of big data workloads. For example, if a convolutional neural network (CNN) is executed on a GPU, a weight matrix and a subset of inputs are assigned to a thread block, where threads within the thread block compute the convolution outputs by sharing the inputs and the weight matrixes through fast on-chip memories. As convolution outputs of different input regions and different weight matrixes do not have any dependencies with each other, it is safe to run them in different thread blocks. GPU hardware architecture is designed to support the hierarchical execution model well. A GPU consists of multiple streaming multiprocessors (which is called SMs in NVIDIA GPU). An SM is comprising with on-chip memories, tens of shader cores, and warp schedulers. A thread block is executed on one of the SMs and each thread is executed on a shader core in the SM. Each SM consists of at least as many shader cores as a half of the warp size. Thus, the threads in each warp execute an instruction in one to two cycles. The mappings and hardware architecture will be explained in detail in section "Hardware Architecture."

Programming Interface

General-purpose GPU (GPGPU) is supported by both hardware architecture and software programming interface. The most popular GPGPU hardware platforms are NVIDIA CUDA and AMD ROCm. Each platform has its software framework, namely CUDA and HIP, respectively. OpenCL is also widely used for GPGPU programming. While CUDA and HIP are specially designed software frameworks for GPGPU, OpenCL is designed for supporting any heterogeneous architectures including GPGPU that typically has a host CPU and an accelerator GPU in the programming model. HIP is designed with a consideration of compatibility with CUDA. Therefore, HIP programs look very similar (or identical) with CUDA programs. HIP programs can run on CUDA architectures by specifying the backend platform as CUDA at compile time.

Code 1 shows a simple vector addition function implemented in OpenCL and CUDA C. Note that HIP code is omitted because it looks alike CUDA code. The overall functional structure is very similar except for some qualifiers and compiler intrinsics. In CUDA, a global thread ID (a unique ID across thread blocks) is calculated by using three compiler intrinsics: *threadIdx*, *blockIdx*, and *blockDim*, each means the thread ID in a thread block, the thread block ID, and the thread block

OpenCL

```
__kernel void vadd (__global const float *a,
                    __global const float *b,
                    __global float *result)
{
    int id = get_global_id (0);
    result[id] = a[id] + b[id];
}
```

CUDA

```
__global__ void vadd (float *a, float *b,
                      float *result)
{
    int id = blockIdx.x * blockDim.x +
                          threadIdx.x;
    result[id] = a[id] + b[id];
}
```

Code 1 Vector addition kernel functions in OpenCL and CUDA

dimension, respectively. On the other hand, OpenCL provides a function that returns the global ID, *get_global_id()*. The kernel function is defined with *__global__* in CUDA and HIP, and *__kernel* in OpenCL. Input and output parameters in CUDA are allocated in global memory (which is the main memory attached in the GPU card) by default and hence there is no need to specify the memory name in CUDA. On the other hand, in OpenCL code, the function parameters need specifications, __global to indicate that the parameters are allocated in global memory. Other than that, the code for the vector addition is identical. Once a thread acquires its global ID, the thread retrieves one element from each of the two input vectors with the ID and adds them into an entry in the output vector. As all threads involved in this GPU program run the same kernel code, *N*-element vector additions can be done in parallel with *N* threads with only two code lines without any loop iterations.

GPUs can be integrated into a system either through PCI-E bus or on the same processor package with the host CPU. If GPUs are connected through PCI-E bus, GPUs typically have their dedicated memory. Therefore, input and output data need to be explicitly sent from/to the CPU system memory. Also, the memory space for these data is required to be allocated via GPU APIs. Code 2 shows the program that is executed on the host CPU to run the vector addition function. In this example code, 32-element arrays A and B are passed to the GPU kernel and added to an output 32-element array C. The input arrays A and B are allocated in the GPU memory with *cudaMalloc* and the array values are sent to the GPU via *cudaMemcpy*. The kernel is invoked with a thread block of 32 threads, as defined in variables numBlocks and numThreads. These thread and block information are passed to the kernel between <<< and >>> marks. The pointers for the input and output data are passed as regular CPU functions. Once the kernel is finished, the output data needs to be copied back to the CPU system memory as can be seen in the last line of the code. HIP-equivalent program can be written by replacing cudaMalloc

```
int main()
{   // executed on CPU
   int A[32] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, … 31};
   int B[32] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0, … 2};
   int C[32] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, … 0};

   int *g_A, *g_B, *g_C;
   cudaMalloc((void**)&g_A, 32 * sizeof(int));
   cudaMalloc((void**)&g_B, 32 * sizeof(int));
   cudaMalloc((void**)&g_C, 32 * sizeof(int));
   cudaMemcpy(g_A, A, 32 * sizeof(int),cudaMemcpyHostToDevice);
   cudaMemcpy(g_B, B, 32 * sizeof(int),cudaMemcpyHostToDevice);

   int numBlocks = 1;
   int numThreads = 32;
   vadd <<<numBlock,numThreads>>>(g_A,g_B,g_C);
   cudaMemcpy(C, g_C, 32 * sizeof(int),cudaMemcpyDeviceToHost);
}
```

Code 2 Host program for the vector addition kernel in CUDA

with *hipMalloc*, cudaMemcpy with *hipMemcpy*, and the kernel invocation line with *hipLaunchKernel*. As the focus of this chapter is not GPGPU programming, descriptions for more APIs and compiler intrinsics are omitted. The readers can find the up-to-date programming interfaces from CUDA programming guide (NVIDIA 2022) and HIP programming guide (AMD 2021).

Hardware Architecture

Since introduced in mid-2000s, GPGPU architectures have been revised at every new generation to improve performance and energy efficiency. However, the core components such as the hierarchical structure of streaming multiprocessors (SMs) and shader cores, as well as the types of memories have not been significantly changed. Therefore, this section will focus on exploring the core components with brief descriptions of different generations. Descriptions will use NVIDIA CUDA platform as a base architecture while limiting the explanations about vendor-specific features.

Figure 6 is internal architecture of a streaming multiprocessor (SM) of a CUDA Pascal architecture. There are multiple SMs in one GPU. In an SM, there are multiple groups of shader cores (*Core* in the figure) that can execute either an integer or a single-precision floating-point operation each cycle. Memory operations are handled by load/store units (*LDST* in the figure). The LDST units have a *memory coalescing unit* to reduce memory traffic by combining multiple requests that are mapped to the same 32-, 64-, or 128-byte address range into one request. Other complex operations such as cosine, sine, reciprocal, and square root are executed on special function units (*SFU* in the figure). Some GPUs like P100 have double-precision floating-point units (*DP Unit* in the figure). Groups of Core, LDST, and SFU are the core computing components in an SM and make a physical unit to run a thread group. An SM can have one or multiple of these computing units. In P100 architecture as can be seen in Fig. 6, two thread groups can be executed in parallel as there are two identical computing units. Each computing unit has a register file, warp scheduler(s), and instruction buffer(s). Warp scheduler selects one of the ready warps to execute each cycle. Instruction buffer has one to two slots per warp such that any ready warp can be issued with the instruction retrieved from the buffer. Register file is private resource per thread to maintain program context of individual threads. To reduce context switching overhead, GPU register files maintain register spaces for tens of warps. Therefore, the GPU register file size is typically several hundred KBs per SM. In P100 shown in Fig. 6, the aggregated register file size in an SM is 256 KB. More details of these core architectural components of an SM will be explored in the following subsections.

Shader Pipeline

Figure 7 illustrates the graphical view of GPU architecture with device memories and the shader pipeline. Each SM consists of compute cores and memory that are required for SIMT execution. Instructions are executed in warp unit where each

Fig. 6 SM in a CUDA platform, P100 (NVIDIA 2016)

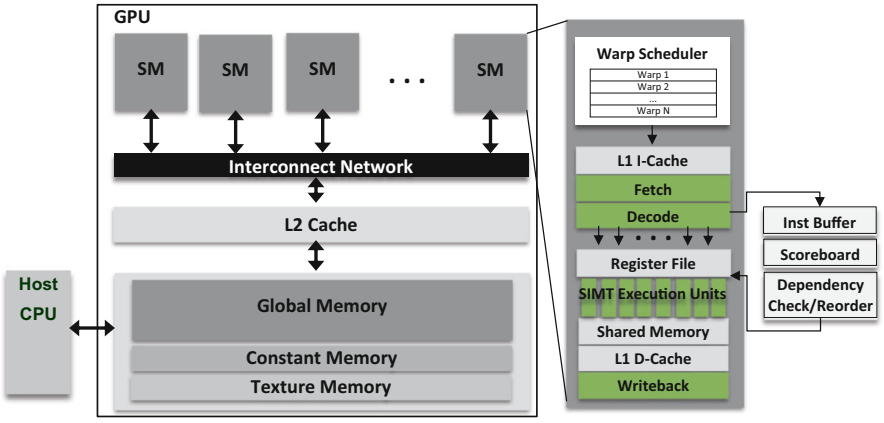


Fig. 7 GPU architecture and pipeline

SIMT lane (thread) is executed on one of the shader cores (or SIMT execution units). A simple in-order pipeline is used that consists of fetch, decode, issue, execute, and writeback steps, as highlighted with green color. After the decode stage, the decoded instructions are enqueued in an *instruction buffer*. The instructions are marked as ready by a *scoreboard logic* if data hazards (e.g., read-after-write and

write-after-write dependencies) are resolved. The scoreboard logic monitors data hazards between instructions by tracking the instructions that reserved registers at decode stage. Once an instruction finishes writeback stage, the reserved register is released, and the scoreboard logic clears the hazards associated to the register. A warp scheduler chooses one or more warps that have ready instructions each cycle. In CUDA architecture, each warp scheduler is associated with multiple dispatch units, and each can issue one ready warp. Therefore, each warp scheduler can schedule as many ready warps as the number of dispatch units. The selected warps issue instructions to the SIMT execution units.

Register File

A CPU normally has a small register file to maintain one or a handful of process contexts. Thus, the register file size is typically up to several hundred bytes. Once a process is scheduled out, context switching is happening by copying current process' register values in the register file to stack memory and moving the new process' register values from stack to the register file. This memory copy-based context switching is used in CPUs because the size of data copy is only several hundred bytes, which induces acceptable performance overhead. However, such a memory copy-based context switching is not practical in GPUs because GPUs run instructions in a warp unit, which consists of 32 threads. To make things worse, the warp-level context switch happens every cycle to hide memory access latency (details will be discussed in section "Warp Scheduler"). This means that 32-thread-worth register copy should be done every cycle, which will cause too much performance overhead. Instead, GPUs employ a large register file that can hold registers of tens of warps. With the large register file, warps can retrieve their registers without any memory copy. In a P100 architecture shown in Fig. 6, each SM has two 128 KB register files. The aggregated register file size in a P100 that has 56 SMs is 14 MB. The register file space is flexibly utilized by active warps. For example, a P100 architecture allows up to 255 registers per thread. As each register file in an SM contains 32,768 32-bit registers (as noted in Fig. 6), up to four warps can be supported when threads use all 255 registers ($32{,}768 \approx 32$ threads/warp $\times$ 255 registers/thread $\times$ 4 warps). Or, if threads use only 32 registers for a given kernel, up to 32 active warps can be supported.

To feed all threads in a warp in each cycle, GPUs employ a banked register file. Figure 8 illustrates the banked register file. Each bank can feed multiple threads per warp. For example, if a bank has 128-byte width, each entry of a bank can hold four 32-bit registers such as r1 for thread 0 to thread 3 of the same warp. In other words, a warp accesses eight banks to retrieve r1s of all 32 threads in a cycle. Though each register can be read in one cycle, each instruction typically consumes multiple cycles to read all operands used by each instruction. Until when all operand values of an instruction are collected, the retrieved registers are buffered in *operand collector unit*. Each warp has up to three slots in the operand collector unit because GPU instructions can use up to three operands. Each slot consists of 32 32-bit entries to hold registers of all 32 threads in a warp. Once all operand values are retrieved, the instruction is issued to the corresponding execution units.

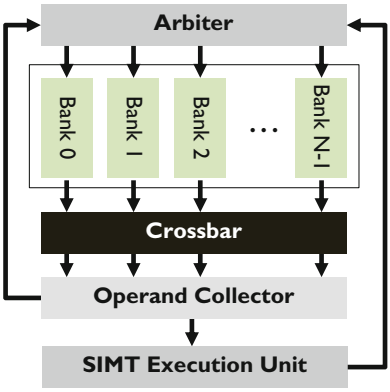


Fig. 8 Banked register file

Warp Scheduler

In GPU, instructions are executed in a warp unit. The threads in a warp execute the same instruction every cycle in a lock-step manner by sharing PC values. This warp-level execution enables batched processing of data intensive workloads. However, as GPU uses in-order execution, once a warp encounters stalls due to data hazards or structural hazards (e.g., bank conflicts in register file or shared memory accesses), the warp should stop execution until the hazard is resolved. Therefore, GPU schedules different warp every cycle. It is known that data hazard stalls can be hidden with at least six warps. To select different ready warps each cycle, GPUs are equipped with warp schedulers. The warp scheduler checks the availability of each warp and selects one ready warp to issue an instruction. Round-robin is the baseline scheduling algorithm that chooses warps based on the warp ID in either increasing or decreasing order. Round-robin is a simple and fair scheduling algorithm, but it is not effective to hide memory stall latency. As threads within a warp execute the same instruction, once a memory load operation is executed, 32-thread-worth data should be loaded from memory together, which takes several hundred cycles. To make things worse, as warps run the same kernel code while scheduled interleavingly in a round-robin scheduling, the neighboring warps are likely to execute the same instruction. Therefore, once a warp encounters a memory stall, following a few warps are likely to encounter the same memory stalls. This makes most of the warps to stuck at memory stalls, which leads the compute cores to be idle for a long time.

To resolve the limitations of round-robin scheduler, several warp schedulers have been proposed. One representative scheduler is two-level scheduler that is proposed by NVIDIA researchers (Gebhart et al. 2011). The two-level scheduler uses two warp queues, one is for ready warps and another for pending warps. Only a small number of ready warps are enqueued in the ready queue and the remaining ready warps and the warps stuck at memory stalls are enqueued to the pending queue. Each cycle, the warp scheduler chooses one in the ready queue in round-robin fashion. Once a warp in the ready queue encounters a memory stall, the warp is moved to the pending queue and one ready warp in the pending queue is enqueued to the ready queue. By running a small group of warps before the other warps, the two-level

scheduler effectively increases the gap of execution timing of memory instructions among the warps. Therefore, the probability of having all warps to stuck at memory stalls at the same time window becomes low. Greedy-then-oldest (GTO) is a more aggressive scheduler that allows one warp to continue the execution until when it encounters stalls and then switches to the oldest ready warp (Rogers et al. 2012). By sacrificing fairness, GTO maximizes the memory access timing gap among the warps. Therefore, there is even less chance that multiple warps encounter memory stalls at the same time. More warp schedulers will be discussed in section "Recent Research on GPU Architecture."

SIMT Stack

Threads in a warp execute the same instruction every cycle, by sharing a program counter (PC) value. For non-branch operations, this warp-level execution is easy to handle. Every cycle, warp PC value is incremented together and proceed to the next line of code. But for the control branches, threads in a warp may need to execute different diverged flows. For example, for an if-else statement in Fig. 9, the even numbered threads should execute the if clause while the odd numbered threads must execute the else clause.

In this case, GPUs do not have capability to execute both if and else clauses in parallel because threads in a warp share the same PC value. Instead, each warp traverses both clauses sequentially. To allow threads to execute the correct diverged flow only (either if or else clause in this example), GPU uses *active mask*. An active mask is a 32-bit vector that indicates the activeness of individual threads in a warp. If a thread has a value "1" in its entry in the active mask, the thread can execute an instruction. If otherwise, the thread skips execution that cycle. In Fig. 9, the right-hand-side flow graph shows the active mask of each path. The if statement (①) is executed by all 32 threads, so the active mask has all 1's. The if clause (②) is entered by even numbered threads so the active mask has 1's for every other bit from the first bit (bits 0, 2, 4, ...). Likely else clause (③) has 1's for every other bit from the second bit (bits 1, 3, 5, ...). At the end of the if-else statement (④), the threads converge again so the active mask is fully filled with 32 1's.

The active mask value is updated every cycle to enforce correct execution. To enable threads in a warp to execute all diverged flows and converge at their immediate post-dominator reconvergence point, GPU employs a SIMT stack.

```
①if (threadId.x % 2 == 0) {
②       // executed by even numbered
        // threads (thread 0, 2, 4, …, 30)
③} else {
        // executed by odd numbered
④       // threads (thread 1, 3, 5, …, 31)
  }
  ... // reconvergence point
```

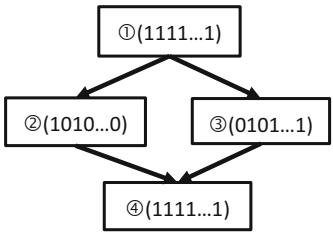


Fig. 9 A diverged flow example (left) and active masks per path (right)

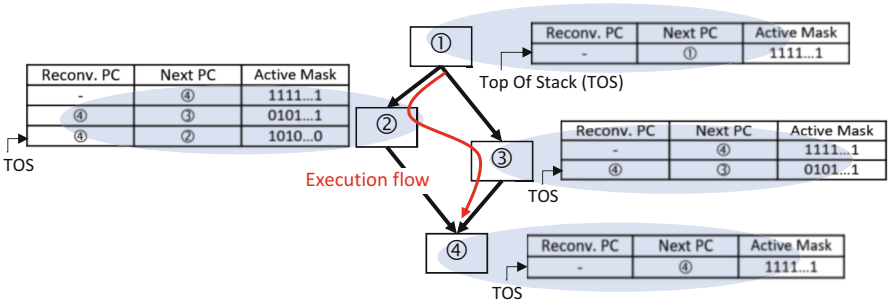


Fig. 10 SIMT stack value transition for the diverged flow example in Fig. 9

A SIMT stack keeps track of the next PC value with the corresponding active mask, as shown in Fig. 10. Once a warp encounters a control branch, the stack grows by as many entries as the number of diverged flows. Each entry has the PC value associated to the diverged flow and the PC value of the reconvergence point. The reconvergence point is the address of the instruction that is the immediate post-dominator (PDOM) of the diverged flows. For example, line ④ is the PDOM of the paths ② and ③ in Fig. 9. If each diverged flow has nested diverged flows, the stack grows further so that all paths can be explored properly. Figure 10 shows the SIMT stack operations while traversing diverged flows ② and ③ sequentially. Once ① find two flows, the stack is grown to hold the two paths as shown with ②. The top of stack (TOS) points to the instructions of path ②. The paths of another flow ③ and the converging point ④ are stacked above them to be executed in the following cycles. After finishing the execution of path ②, the stack is shortened, and TOS points the path of ③. Finally, when the path ③ finishes and the next PC is reaching to that of ④, the stack size is reduced to one entry, which means that there is no more divergence.

Memories

Unlike a CPU that typically uses a large system memory and multilevel caches, a GPU is equipped with several different types of memories where some of them can be accessed with special software APIs and compiler intrinsics.

Figure 7 shows the memory structures of GPU architecture. The following subsections will explain each of the memories.

Global Memory

The GPU-side main memory is called *global memory*. Global memory is the largest read-write region in the GPU device memory and is accessible by the host CPU to send input data and receive output data of a GPU kernel function. The host CPU can use cudaMalloc and cudaMemcpy APIs to allocate variables and send/receive

data in the global memory, as described in section "Programming Interface." Global memory is an off-chip memory that takes hundreds of cycles to access from GPU cores. Though multiple memory controllers are employed to provide high bandwidth, the global memory accesses are one of the main performance bottlenecks. To improve the global memory access performance, each SM has a *memory coalescer*, which collects memory accesses and merges a few requests that map to the same address regions that are aligned with 32-, 64-, and 128-byte units. Through the memory coalescer, a few memory accesses can be served via one memory transaction and hence effectively reduce the demands for memory bandwidth.

Constant Memory and Texture Memory

Constant memory is a read-only data region in GPU device memory. Besides the data that are identified by the compiler as read-only, some read-only data that the programmer defines with *__constant__* compiler directive can be allocated in the constant memory. The values of the programmer-specified constant variables should be filled with a special API, *cudaMemcpyToSymbol()* in CUDA. Constant memory is a cacheable memory (cached in a small constant cache which is <10 KB) and has a broadcasting feature. If all threads within a warp accesses the same data word from the constant memory, the data word is broadcasted to all threads. Therefore, the variables that are accessed commonly by all threads within a warp or those that are small enough to be cached in the constant cache fit well with constant memory. Texture memory is similar to constant memory but used only for texture fetch that is a graphics operation. Texture memory is accessible via special APIs such as *tex1Dfetch* and *tex1DGrad* in CUDA. The accesses to the texture memory are cached in a texture cache.

Shared Memory

Shared memory is an on-chip SRAM memory. One shared memory is utilized by all thread blocks assigned to the same SM. As inter-thread block communication is not supported, shared memory is also logically partitioned for each thread block. Therefore, GPU compiler (e.g., nvcc) checks the aggregated shared memory usage of all thread blocks that are going to be assigned to each SM and generates a compile error if the usage exceeds the shared memory size in an SM. The shared memory is similar to scratch memory in CPU, which is a programmer-controllable on-chip memory. The data stored in shared memory can be accessed with similar latency to L1 cache, while being kept in the shared memory throughout the application's execution time without concern about eviction, unlike L1 cache. With this performance advantage, allocating proper variables in the shared memory is one of the most important performance optimizations in GPU programming. To allocate data in the shared memory, a qualifier *__shared__* should be used. To assign values to the variables defined in the shared memory, regular store operations can be used. For example, to load input data to the shared memory, the input data should be loaded from the global memory to registers and then stored to the shared memory. Similar to the register file, shared memory uses a banked structure. For example,

the shared memory in an NVIDIA Maxwell architecture has 32 banks, where each bank has 4-byte width. If all threads in a warp access different bank, all data will be retrieved in one shared memory access latency (which is around five cycles). If any threads access the same bank but for different data words, there will be a bank conflict and the conflicted accesses will be sequentially executed. Therefore, the shared memory accesses should be carefully designed by the programmer.

L1 and L2 Caches

GPUs typically use two-level cache hierarchy. Each SM has an on-chip L1 cache. In a GPU device, there is a shared L2 cache. L2 cache is connected to the SMs through an interconnection network. Therefore, access latency to L2 is normally much longer than the L1 cache access time (which is around five cycles). Accesses to the global memory is cached in L1 and L2 caches. Unlike CPUs where L1 caches are activated by default, GPU L1 caches can be configured for activation. Programmers can enable L1 cache for a program with a compiler option at compile time. Once L1 is disabled, global memory accesses are directly forwarded to L2, and the L1's physical SRAM space is used for either shared memory or texture cache depending on the GPU architecture configuration.

There can be multiple reasons to disable L1 cache. The two main reasons area the unique characteristics of GPU applications and efficient cache coherence support. Though GPUs are also used for general-purpose applications, the dominant GPU applications are graphics applications. In a graphics application, data locality and reuse rate are much lower than general-purpose applications. Note that in image processing, the pixel values are read, processed, and then stored to the output framebuffer. In this course of processing, the same values are not likely to be repeatedly read. Therefore, L1 cache does not help for performance improvement. Regarding cache coherence, GPUs barely support L1-cache level coherence because thread blocks are regarded to be independent. This means that there is no need to enforce coherent data accesses. But, if an algorithm requires some data sharing across thread blocks (that run on different SMs), the shared L2 is used for coherent data sharing. In this case, the updated data should be flushed to L2 so that all the other SMs can see the up-to-date data. This GPU's two-level coherence is called *scoped* coherence (Hower et al. 2014). If the coherence scope is thread block, threads within the thread block are guaranteed to see the updates made within the thread block, which matches the first case that there is no need to have coherence across L1 caches. If the coherence scope is GPU device, the coherent accesses are forced through cache flushes to L2, which is the second case that all L1 updates are flushed to L2. When GPU-scoped coherence is used, L1 can be disabled because L1 will be bypassed anyways.

Optimization Use Case: Access-Aware Variable Mapping to Memory

In this subsection, an optimization example on a CUDA program will be explored that allocate variables to the memory that supports each variable's access patterns the best. Code 3b is a matrix multiply code in CUDA that is equivalent to the C version code shown in Code 3a. In this naively translated code, all three matrixes

```
#define WIDTH 32
void MatrixMul(float* A, float* B, float* C)
{
      for (i = 0; i < WIDTH; i++)
            for (j = 0; j < WIDTH; j++)
                  for (k = 0; k < WIDTH; k++)
                      C[i * WIDTH + j] += A[i * WIDTH + k] * B[k * WIDTH + j];
}
```

(a)

```
__global__ void MatrixMul(float* A, float* B, float* C)
{
      int tid = threadIdx.x;
      int row = tid / WIDTH;
      int col = tid % WIDTH;
      for (k = 0; k < WIDTH; k++)
            C[row * WIDTH + col] += A[row * WIDTH + k] x B[k * WIDTH + col];
}

int main()
{
      ...
      float *g_A, *g_B, *g_C;
      cudaMalloc((void**)&g_A, WIDTH*WIDTH * sizeof(int));
      cudaMalloc((void**)&g_B, WIDTH*WIDTH * sizeof(int));
      cudaMalloc((void**)&g_C, WIDTH*WIDTH * sizeof(int));

      cudaMemcpy(g_A, A, WIDTH*WIDTH * sizeof(float),cudaMemcpyHostToDevice);
      cudaMemcpy(g_B, B, WIDTH*WIDTH * sizeof(float),cudaMemcpyHostToDevice);
      MatrixMul <<<1,WIDTH*WIDTH>>>(g_A, g_B, g_C)
      ...
}
```

(b)

Code 3 Matrix multiplication in (**a**) C and (**b**) CUDA

are allocated in the global memory. Each thread computes one output matrix element by multiplying one row of A matrix and a column of B matrix.

Let us check the access patterns to these matrixes to find the best memory to map them. The WIDTH value is 32. The kernel is executed by a thread group of 32×32 threads. According to the row and the column calculations in the kernel, the 32 continuous threads in each warp use the same *row* value and consecutive *column* values. In the A matrix index calculation (`A[row * WIDTH + k]`), *row* is the only thread-dependent variable. As the threads within a warp use the same *row* value, each warp accesses the same A matrix element every cycle. Therefore, a memory that is cacheable and can broadcast one value to all threads in a warp is proper for the matrix A, which is the constant memory. On the other hand, regarding B matrix index calculation (`B[k * WIDTH + col]`), all threads in a warp access different elements in the same row. Therefore, constant memory is not proper. A memory that allows parallel accesses such as banked memory structure is proper. The shared memory is a fast on-chip memory that has banked structure. Therefore, shared memory can well support the matrix B accesses. Finally, regarding the C matrix, each element in C matrix is independently calculated by one thread. This means that threads can use a fast private memory for the computation.

```
__constant__ float c_A [WIDTH*WIDTH];                  ----------------------   ①
__global__ void MatrixMul(float* B, float* C)          ----------------------   ②
{
      int tid = threadIdx.x;
      int row = tid/WIDTH;
      int col = tid%WIDTH;
      float lC = 0.0;                                  ----------------------   ③

      __shared__ float sB[WIDTH][WIDTH];               ----------------------   ④
      sB[row][col] = B[row * WIDTH + col];             ----------------------   ⑤
      __syncthreads();                                 ----------------------   ⑥

      for (k = 0; k < WIDTH; k++)
            lC += c_A[row * WIDTH + k] * sB[k][col]; ----------------------   ⑦

      C[row * WIDTH + col] = lC;                       ----------------------   ⑧
}
int main()
{
      ...
      float *g_B, *g_C;
      cudaMalloc((void**)&g_B, WIDTH*WIDTH * sizeof(int));
      cudaMemcpyToSymbol(c_A, A, WIDTH*WIDTH * sizeof(float)); -------------   ⑨
      cudaMemcpy(g_B, B, WIDTH*WIDTH * sizeof(float),cudaMemcpyHostToDevice);

      MatrixMul <<<numBlock,numThreads>>>(g_B,g_C);
      ...
 }
```

Code 4 Matrix multiplication in CUDA with memory assignment optimizations: matrix A in constant memory, matrix B in shared memory, and matrix C in register

Only when all the computations are finished, the result can be updated to the shared C matrix which is in the global memory. Therefore, the register file, which is the fastest private memory space in GPU architecture, is optimal for matrix C computation.

Code 4 shows the optimized CUDA code. The A matrix is defined as a global variable with a qualifier *__constant__* as shown in line ①. Then, an API, *cudaMemcpyToSymbol*() is used to copy matrix contents from CPU system memory to the constant memory (⑨). As the matrix A is defined as a global variable, the matrix A does not need to be passed as an input parameter (②). To define B matrix in the shared memory, a qualifier *__shared__* is used in the kernel (④). As shared memory is not directly accessible by the host CPU, the matrix B is initially copied to the GPU global memory. Then, the matrix contents should be copied from the global memory to the shared memory (⑤). The line ⑤ makes all threads to collaboratively load matrix contents from the global memory to the shared memory, one element per thread. To wait until all threads to finish data copy, a synchronization function, *__syncthreads()*, is called (⑥). To maintain the intermediate computation result of each thread in a register, a local variable is defined (③). If a variable is defined in a kernel, each thread has one register entry for that variable. ⑦ shows the matrix multiplication, where each thread multiples a row of A matrix with a column of B matrix. The computation result is collaboratively copied to the output matrix, C in the global memory, one element per thread (⑧).

Recent Research on GPU Architecture

Performance

GPGPU performance has been ever increasing since released in mid-2000s thanks to the significant efforts made by both academia and industry. In this subsection, a few selected research will be introduced that tackled memory access latency, which is one of the most critical performance bottlenecks in GPU computing.

Hiding Memory Access Latency with Advanced Warp Schedulers

Warp schedulers can improve performance by effectively hiding various stalls through different scheduling orders. One of the most performance critical stalls is memory stall. As described in section "Warp Scheduler," two-level scheduler and GTO are also designed to mitigate long memory access latency. While the two-level scheduler and GTO improved performance by enlarging the memory operation execution timing gap among warps, the following studies suggested more proactive approaches to improve data locality thereby fundamentally reducing the number of memory accesses.

Cache conscious wavefront scheduler (Rogers et al. 2012) reorders warps to exploit data locality in L1 cache as much as possible. Their wavefront scheduler uses locality scoring information that takes L1 cache locality loss into account. To track the locality loss, they employ a small victim tag array (VTA) that is accessed for each L1 cache misses. If a wavefront has higher VTA hits (which means that the wavefront has higher locality loss), the wavefront is excluded from scheduling to reduce memory stalls.

While cache conscious wavefront scheduler operated passively, divergence-aware warp scheduler (Rogers et al. 2013) proactively predicts cache usage and schedules wavefronts such that the data can be reused in L1 cache without exceeding the cache capacity. The authors observed that cache locality is reduced while a warp traverses diverged flows. If a new warp's execution is predicted to evict earlier warp's cached data that needs to be reused for the diverged another flow, the scheduler excludes the new warp from the scheduling such that the earlier warp can finish the execution without losing cache locality.

Sethia et al. (2015) similarly exploited locality but focused on tackling memory resource contention caused by requests issued by multiple warps. Their memory-aware scheduling prioritizes one warp's memory requests over all other warps when memory resource is saturated to finish the warp's execution as soon as possible, thereby increasing the possibility of compute and memory request overlapping. To maximize the impact of the memory-aware scheduling, they also proposed cache access re-execution that moves warps to a re-execution queue if the warps are stalled at load-store unit due to memory backpressure such that the other warps that may have cache hit can be served earlier.

Some studies considered warp criticality in warp scheduling to improve fairness and performance (Lee et al. 2015b). The criticality-aware schedulers tackle execu-

tion time imbalance across warps and improve performance by allocating more time resources to the warps in starvation. To identify critical warp, a criticality prediction logic monitors the execution progress of individual warps by integrating one criticality counter per warp. The criticality counter is updated by using instruction count disparity caused by diverged flows and the stall latencies caused by shared resource contentions.

Some other studies aimed at finding a warp scheduling algorithm that can improve the effectiveness of memory prefetch. Jog et al. (2013) presented a prefetch-aware warp scheduling policy. They observed that the simple next-line prefetcher cannot improve performance as expected under two-level or round-robin warp schedulers because the warps that use the prefetched next line is likely to be scheduled in the immediate-following scheduling cycle. Note that warps in a thread group typically access consecutive memory addresses and hence the next cache line prefetched by a warp is consumed by an immediate-following warp. Because immediate-following warp is scheduled one scheduling cycle later than the warp that issued a prefetch, the prefetched data are likely to arrive too late. To place an enough time gap between prefetch and the data access, the scheduler forms fetch groups with nonconsecutive warps. By scheduling the warps in the same fetch group in the consecutive time windows and having the warps in the group issue prefetch for another fetch group, prefetch can be issued way ahead of time than the actual accesses.

Throttling Memory Access Latency

While the earlier subsection explored studies that used warp scheduling for hiding memory stall cycles, this subsection will introduce research that utilize on-chip memory to throttle the off-chip memory access latency.

Kim et al. (2016) proposed to utilize memory stall times to pre-execute valuable computations. The authors argue against the conventional wisdom that more active warps can hide memory latency because more warps may increase on-chip memory contention. Instead, they allow warps to continue fetch and decode successive instructions when encountering long-memory stalls. If the instructions are turned out to be independent on the stalled memory operation, the instructions can be even pre-executed. To avoid data hazards caused by this pre-execution, the pre-executed outcomes are renamed to a new physical register. Once the stall is cleared, the warp can simply use the renamed pre-executed results without executing the instructions again.

Koo et al. (2017) used a pilot warp to detect access pattern of individual data and apply different cache management approach for each access type. For example, streaming (access once and use once) data are better to bypass cache to avoid unnecessary cache thrashing. The data having intra-warp locality (used multiple times by a warp) or inter-warp locality (used multiple times by multiple warps) are pinned to the cache until when the data is accessed by all requesting warps. With the access-aware cache management, they improved locality in on-chip caches.

Oh et al. (2019) extended on-chip cache space to underutilized register space to improve data locality. The proposed linebacker cache management utilizes

underutilized register file space as a victim cache. Once a data in the victim cache is accessed again, the data can be quickly copied to the accessed instructions inside the register file. To improve the opportunities to spare victim cache space in the register file, they apply CPU-like thread group context switching, where the register values of an inactive thread group is copied to the off-chip memory. Using register space as a victim cache can have better performance than the conventional victim cache because data can be moved via an intra-register file copy when reused.

Some studies reduced memory traffic by increasing the opportunities of intra-register data sharing. The following two studies exploited unique computation patterns of deep learning workloads for intra-register data sharing. Jeon et al. (2019) observed high data sharing opportunities among neighboring neurons in the convolution operations in the convolutional neural network (CNN). The proposed perfect sharing and partial sharing enable the neurons to access data from register file if the data are fetched by neighboring neurons already, instead of redundant memory accesses. To enable zero-copy register-level data sharing, they proposed to simply rename the physical register pointer with the architected register of the requesting neuron (thread). Kim et al. (2020) similarly exploited high data locality in convolution operations with register-level data sharing. They track the history of memory accesses to find the existence of a data in the register file. Once data is located in the register file, they map the data to the requested warp by using register renaming. Their design accommodates the unique access patterns made by CUDA Tensor Core operations.

A recent study proposed to make the L1 caches as a shared resource for multiple SMs. Ibrahim et al. (2021) observed that local L1 caches are not efficiently utilized because data that are commonly used by thread blocks are redundantly loaded to multiple L1 caches. Also, the bandwidth imbalance between local L1 and the common L2 is significant. To resolve these problems, they designed L1 caches to be decoupled to SMs and interfaced with the SMs via interconnect network. Depending on the access patterns, the connections are aggregated or private. They explored performance impacts of various mapping configurations.

Energy Efficiency

GPUs show better energy efficiency (in FLOPs per watts) than CPUs because GPUs can achieve better throughput per watt with massive parallelism even without using fancy architecture components such as complex branch predictor, out-of-order execution, and cache coherence protocols, unlike CPUs. However, due to abundant computing resources that enable massive parallelism (e.g., hundreds of compute cores and megabytes of register file), the overall power consumption increases rapidly for each new GPU generation. With the increasing power consumption, it is hard to integrate more computing logics that are essential for improving performance. Thus, the power overhead will eventually slow down the performance improvement of GPUs. There have been extensive efforts from both academia and industry to improve GPU energy efficiency. Figure 11 shows architecture

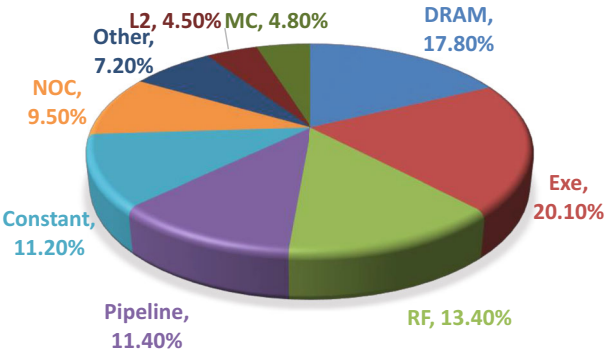


Fig. 11 Component energy breakdown for GTX480

Fig. 12 Modified pipeline with scheduling information in NVIDIA Kepler architecture

component-level energy breakdown of a GPU device (Abdel-Majeed et al. 2017). The compute cores, DRAM, register file, and pipeline are the top power consumers so this section will explore the research that tackled the energy efficiency of these components.

Revisiting Compute Cores and Pipeline

There have been efforts driven by the industry. For example, NVIDIA began considering power efficiency as one of the most important design requirements from their Kepler architecture. The most significant effort made for the Kepler architecture is a simplified warp scheduling. As discussed in section "Shader Pipeline," data hazard is detected at run time by using a scoreboard before Kepler architecture. But, as GPUs use in-order execution with simple pipeline steps, the timing when individual operands become ready is easy to estimate even at compile time at least for the non-memory operations. By leveraging such a deterministic scheduling timing, they revised the compiler to generate the instruction ready timing (NVIDIA 2012). NVIDIA did not disclose the format of the scheduling information, but some studies reverse-engineered and found that a scheduling information is embedded per seven instructions in a similar format with the explicit dependence look-ahead instruction used in Tera computer system (Lai and Seznec 2013; Alverson et al. 1990). As illustrated in Fig. 12, the timing information is decoded at run time by a newly introduced *sched. Info* pipeline stage and used by the warp scheduler at another new pipeline stage *select* for determining the ready instruction to issue. The statically estimated scheduling information effectively replaces the complex hardware component for dependency check and reduces power consumption.

Some studies exploited the inherent similarity in operand values and instructions to save power. Wong et al. (2016) leveraged the operand value similarity to reduce energy consumption. Namely warp approximation, the proposed method detects value similarity in the operands and makes one representative SIMT lane to execute instructions and the corresponding register accesses on behalf of multiple lanes that use similar operands. By providing a programming interface that enables programmers to annotate the regions of code that can safely run warp approximation, the warp approximation improves GPU energy efficiency by 26% with negligible final output errors.

Kim and Ro (2018) observed that there are quite a few identical warp-level instructions across thread blocks. While threads in a warp may use similar but not identical operand values, if one goes upper level to an inter-warp level, there are a few warps that use exactly the same operands because thread blocks execute the same kernel code and thread id is repeated across thread blocks. Such a redundant warp execution may lead to energy overhead. Thus, the authors proposed to reuse warp instruction and the corresponding registers. By eliminating redundant executions, dynamic power was effectively reduced. The register reuse also saves register usage by mapping one physical register for multiple architected registers.

Revisiting Register File

As discussed in section "Register File," GPUs have a huge register file to support faster warp context switching. Therefore, the register file becomes the third power hungry component and the most power consuming on-chip memory in GPU architecture. There have been a few studies that tried to shrink the size of register file for better power efficiency. Jeon et al. (2015) pointed that the register file size is overprovisioned because the registers are considered as private resource for each thread. As each thread needs to have its own register space that is not shared with any other threads, the register file size is proportionally increasing as higher parallelism (more active warps) is supported. However, they observed that the register file is significantly underutilized, even for the workloads that has high register file occupancy (almost all register space is occupied by the active warps). The main reason of the high underutilization is the existence of short-lived registers. While one register usage in the compiled binary is mapped to one physical register allocation in CPU, one register in a compiled binary in GPU computing leads to tens to hundreds of register space allocation because all threads that are assigned to the same kernel should have their own register entry. For example, if a kernel has a short-lived register that is dead for most of the kernel execution time and the kernel is executed by 12 warps, there will be 12×32 ($= 384$) short-lived register spaces. These 384 short-lived registers will consume power during the entire execution time while being used for actual computation for a short period of time. Jeon et al. (2015) proposed to make the register file as a shared resource across warps. Their register file virtualization uses compiler-generated register liveness information to map an architected register to an available physical register. The registers are mapped when a warp stores a new value to the architected register and released when the register lifetime is over. With this dynamic register mapping, the GPU can maintain only

the live registers, thereby running most of the GPU workloads with a 50% smaller register file without performance degradation. By cutting the register file by half, the register file virtualization effectively reduces GPU static and dynamic power consumption. They also designed a scheduling algorithm to avoid deadlock situation due to limited register resource for large workloads.

Lee et al. (2015) tackled the register file energy efficiency by shrinking the space requirement of each warp-level register. They observed that GPU workloads have inherent value similarity among threads within a warp. For example, if a warp explores an array, threads in that warp are typically assigned to access consecutive elements. Therefore, neighboring threads typically use consecutive array index values, where the value distance between adjacent threads is only 1. The authors pointed that storing these continuous values to 32 separate register entries is waste of register space. Instead, they integrate a simple base-delta-immediate compression where the first thread of a warp stores a base value in a 32-bit register entry while the remaining 31 threads store only the distance values. This way, the warp-level register space is effectively shrunk up to 2 bytes. The compressed register values are maintained in fewer register banks than the uncompressed ones. The register file compression reduces both dynamic and static power of a register file by reducing the number of register bank accesses and enabling the opportunistic power gating on the empty register banks.

Lee et al. (2017) extended the register file compression to support compressed execution. By leveraging the adder and subtractor logics used by the operand decompressor, they allow uncompressed data additions and subtractions to be done inside the register file. The compressed execution can bypass the execution stage, thereby saving execution unit power consumption.

Esfeden et al. (2019) further increased the register file utilization efficiency by squeezing multiple narrow-width register values into one register entry. The authors observed that a significant amount of register values in GPU computing falls into one to two byes only. To reduce the register bandwidth for retrieving unnecessary bits, they combine multiple architected registers of a warp into one register entry. To avoid potential register bank conflicts caused by the register packing, they applied graph coloring algorithm for finding a pair of registers to be packed. The registers that are commonly used by the same instruction are packed together. This way, the register bandwidth can be even further reduced because two operands of an instruction can be read by one register entry access. With the operand coalescing and packing, both register file usage and accesses are reduced, which lead to improvements of register file power efficiency and overall performance.

Reliability

With Moore's Law, transistor size is decreased to even a several nanometer scale today. Though the small transistor size helps integrate more cores and fancy micro-architecture components on the integrated circuits, it also increases vulnerability to various hardware errors. Most of permanent errors (e.g., stuck-at-zero) can be

screened at the testing steps in the fabrication process. However, soft errors that are caused by particle strikes coming from cosmic ray or processor packages cannot be completely filtered. Also, the denser the transistors are integrated, the more transistors can be struck by one soft error. Thus, the concerns for multi-bit flips are increasing. The non-general-purpose GPUs have not integrated reliability supports because graphics applications are inherently error prone. Note that one to two pixel errors in an image are acceptable if those are not perceivable by human eyes. But, as GPUs are used for general-purpose applications, a few bit flips may lead to a critical computation error. There have been a few studies that try to detect and correct errors occurring in various architecture components in GPUs.

Run-Time Error Detection and Correction

Jeon and Annavaram (2012) and Tan and Fu (2012) detect soft errors occurring in compute cores. Both studies applied dual modular redundancy (DMR) to detect erroneous computations. DMR consists of a core and a checker core. One instruction stream is duplicated and fed to the core and the checker core. If the computation results are different, one of the cores is considered to have errors. If the error is not repeated (not a permanent error), the same instruction stream is fed again to correct the error. DMR has a high error coverage because the entire program execution can be checked every cycle. But it has high area and energy overhead because an extra core (checker core) is needed to verify the computation. Jeon and Annavaram (2012) mitigated the drawbacks of DMR by leveraging many cores already embedded in the GPU. As discussed in section "Hardware Architecture," warp executions may be diverged when encountering conditional branches. GPU handles the divergence by traversing all divergent flows sequentially. While the warp is executing a divergent flow, the threads that do not follow the flow should be disabled and the corresponding cores are idled. To run DMR without adding extra checker cores, they proposed to leverage such cores that are idled by various reasons including warp divergence to check active cores' execution. A register value forwarding unit feeds the same operand values to the active core and the paired idle core. The computation results of the two cores are checked after the execution. On the other hand, Tan and Fu (2012) leveraged stall cycles caused by various resource contentions, especially the long memory latency for redundant execution. While a core is stuck at memory stalls, their proposed design verifies the instruction stream that has not already been verified.

These studies demonstrated a lightweight DMR to detect soft errors on compute cores. However, their error coverage is dependent on program control flows and the detected errors cannot be corrected. Abdel-Majeed et al. (2015) improved the error coverage by leveraging operand value similarity among threads within each warp. Once some threads use the same operand values, they pair the two active threads to verify the computation. The value-similarity based DMR pairing does not require an idle core for verification purposes. If there is not enough pairable cores, they partition a warp into multiple subgroups to proactively make idle cores in each subgroup and verify all active threads' execution. The subgrouping adds performance overhead because a warp execution needs one extra computation cycle

with subgroups. However, this subgrouping enables to implement triple modular redundancy (TMR) that can correct errors. Their extended warp subgrouping allocates three cores to run the same instruction stream, either by assigning two same-valued active cores and an idle core or one active core with two idle cores.

Fault Analysis

The aforementioned studies aimed at detecting and correcting any errors occurred in the computations regardless the actual location of the error. This is called as coarse-grained error coverage. Some other studies examined vulnerability in finer-grained level, such as each logic-gate level, register bit-level, or pipeline stage-level. The finer-grained vulnerability studies use fault injection methods. They pick a few fault sites in either some architecture components or data, inject errors to the fault sites (e.g., flips bits), and check the total number of corruptions and detected errors from an application execution. Nie et al. (2018) examined the necessary fault sites to inject errors to evaluate overall vulnerability in GPU computing. They observed that the abundant resources and massive parallelism of GPU computing require huge number of fault sites which is almost impractical to run without a significant performance overhead. To reduce the overhead, they leveraged inherent computation redundancies among threads, warps, and thread blocks in GPU computing. If there are any redundant (even when the operands are not exactly matching) control flows, they add faults only to some representative executions. For example, if there is a warp divergence, errors are injected to only one thread's execution per diverged flow. Likely, some representative loop iterations for a whole loop execution and a few bits among all bits in each data that are more vulnerable to errors are identified as fault sites. With such a significant fault site reduction, they achieved similar error coverage with non-pruned approach, with significantly less effort for fault injection.

Conclusion

This chapter describes basic concepts and design details of GPU architecture. The focus of this chapter is to help readers to understand the reasons behind the high throughput of GPU computing. As GPU architecture and programming interfaces are quickly evolving, this chapter explores the core architecture components and the interfaces, with which the readers can easily catch up on the advanced features and latest updates of GPU architectures.

References

Abdel-Majeed M, Dweik W, Jeon H, Annavaram M (2015) Warped-RE: low-cost error detection and correction in GPUs. In: Proceedings of the 45th annual IEEE/IFIP international conference on dependable systems and networks, 2015 June 22–25, Rio de Janeiro, Brazil

Abdel-Majeed M, Shafaei A, Jeon H, Pedram M, Annavaram M (2017) Pilot register file: energy efficient partitioned register file for GPUs. In: Proceedings of the IEEE international symposium on High performance computer architecture (HPCA), 2017 Feb 4–8, Austin, TX, USA

Alverson R, Callahan D, Cummings D, Koblenz B, Porterfield A, Smith B (1990) The tera computer system. In: ACM SIGARCH computer architecture news, 1990 Sept, vol 18(3b), pp 1–6

AMD (2021) AMD HIP programming guide v1.0. [Internet]. Available from: https://github.com/RadeonOpenCompute/ROCm/blob/master/AMD_HIP_Programming_Guide.pdf

Esfeden HA, Khorasani F, Jeon H, Wong D, Abu-Ghazaleh NB (2019) CORF: Coalescing Operand Register File for GPUs. In: international conference on architectural support for programming languages and operating systems, April 2019, Providence, RI

Gebhart M, Keckler SW, Dally WJ (2011) A compile-time managed multi-level register file hierarchy. In: Proceedings of the 45th annual IEEE/ACM international symposium on microarchitecture (MICRO), 2011 Dec 3–7, Porto Alegre Brazil

Hower DR, Hechtman BA, Beckmann BM, Gaster BR, Hill MD, Reinhardt SK, Wood DA (2014) Heterogeneous-race-free memory models. In: Proceedings of the international conference on architectural support for programming languages and operating systems (ASPLOS), Mar 1–5 2014, Salt Lake City, Utah, USA

Ibrahim MA, Kayiran O, Eckert Y, Loh GH, Jog A (2021) Analyzing and leveraging decoupled L1 caches in GPUs. In: Proceedings of the IEEE international symposium on high-performance computer architecture (HPCA), Feb 27–Mar 3 2021, Seoul, Korea

Jeon H, Annavaram M (2012) Warped-DMR: light-weight error detection for GPGPU. In: Proceedings of the 45th annual IEEE/ACM international symposium on microarchitecture (MICRO), 2012 Dec 1–5, Vancouver, BC, Canada

Jeon H, Ravi GS, Kim NS, Annavaram M (2015) GPU register file virtualization. In: Proceedings of the 48th annual IEEE/ACM international symposium on microarchitecture (MICRO), 2015 Dec 5–9, Waikiki, HI, USA

Jeon H, Esfeden HA, Abu-Ghazaleh NB, Wong D, Elango S (2019) Locality-aware GPU register file. IEEE Comput Archit Lett 18(2):153–156

Jog A, Kayiran O, Mishra AK, Kandemir MT, Mutlu O, Iyer R, Das CR (2013) Orchestrated scheduling and prefetching for GPGPUs. In: Proceedings of the 40th annual international symposium on computer architecture (ISCA), 2013 June 23, Tel Aviv, Israel

Kim K, Wo RW (2018) WIR: warp instruction reuse to minimize repeated computations in GPUs. In: Proceedings of the IEEE international symposium on High Performance Computer Architecture (HPCA), 2018 Feb 24–28, Vienna, Austria

Kim K, Lee S, Yoon MK, Koo G, Ro WW, Annavaram M (2016) Warped-preexecution: a GPU pre-execution approach for improving latency hiding. In: Proceedings of the IEEE international symposium on high performance computer architecture (HPCA), 2016 Mar 12–16, Barcelona, Spain

Kim H, Ahn S, Oh Y, Bo K, Ro WW, Song W (2020) Duplo: lifting redundant memory accesses of deep neural networks for GPU tensor cores. In: Proceedings of the 53rd annual IEEE/ACM international symposium on microarchitecture (MICRO), 2020 Oct 17–21, Athens, Greece

Koo G, Oh Y, Ro WW, Annavaram M (2017) Access pattern-aware cache management for improving data utilization in GPU. In: Proceedings of the ACM/IEEE 44th annual international symposium on computer architecture (ISCA), 2017 June 24–28, Toronto, ON, Canada

Lai J, Seznec A (2013) Performance upper bound analysis and optimization of SGEMM on Fermi and Kepler GPUs. In: Proceedings of the 2013 IEEE/ACM international symposium on code generation and optimization (CGO), 2013 Feb 23, pp 1–10

Lee S, Kim K, Koo G, Jeon H, Ro WW, Annavaram M (2015) Warped-compression: enabling power efficient GPUs through register compression. In: Proceedings of the ACM/IEEE 42nd annual international symposium on computer architecture (ISCA), 2015 June 13–17, Portland, OR, USA

Lee S, Arunkumar A, Wu C (2015b) CAWA: coordinated warp scheduling and cache prioritization for critical warp acceleration of GPGPU workloads. In: Proceedings of the ACM/IEEE 42nd annual international symposium on computer architecture (ISCA), 2015 June 13–17, Portland, OR, USA

Lee S, Kim K, Koo G, Jeon H, Annavaram M, Ro WW (2017) Improving energy efficiency of GPUs through data compression and compressed execution. IEEE Trans Comp 66(5):834–847

Nie B, Yang L, Jog A, Smirni E (2018) Fault site pruning for practical reliability analysis of GPGPU applications. In: Proceedings of the 51st international symposium on microarchitecture (MICRO), 2018 Oct 20–24, Fukuoka, Japan

NVIDIA (2012) NVIDIA Geforce GTX 680 white paper v1.0. [Internet]. Available from: https://www.nvidia.com/content/PDF/product-specifications/GeForce_GTX_680_Whitepaper_FINAL.pdf

NVIDIA (2016) NVIDIA Tesla P100 white paper v1.1. [Internet]. Available from: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

NVIDIA (2022) CUDA C++ Programming Guide v11.6. [Internet]. Available from: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

Oh Y, Koo G, Annavaram M, Ro WW (2019) Linebacker: preserving victim cache lines in idle register files of GPUs. In: Proceedings of the ACM/IEEE 46th annual international symposium on computer architecture (ISCA), 2019 June 22–26, Phoenix, AZ, USA

Pattnaik A, Tang X, Kayiran O, Jog A, Mishra A, Kandemir MT, Sivasubramaniam A, Das CR (2019) Opportunistic computing in GPU architectures. In: Proceedings of the 46th international symposium on computer architecture (ISCA), 2019 June 22, Phoenix, Arizona

Rogers TG, O'Connor M, Aamodt TM (2012) Cache-conscious wavefront scheduling. In: Proceedings of the IEEE/ACM 45th annual international symposium on microarchitecture (MICRO), 2012 Dec 1–5, Vancouver, BC, Canada

Rogers TG, O'Connor M, Aamodt TM (2013) Divergence-aware warp scheduling. In: Proceedings of the IEEE/ACM 45th annual international symposium on microarchitecture (MICRO), 2013 Dec 7–11, Davis, CA, USA

Sethia A, Jamshidi D A, Mahlke S (2015) Mascar: speeding up GPU warps by reducing memory pitstops. In: IEEE 21st international symposium on high performance computer architecture (HPCA), 2015 Feb 7–11, Burlingame, CA, USA

Tan J, Fu X (2012) RISE: improving the streaming processors reliability against soft errors in GPGPUs. In: Proceedings of the 21st international conference on parallel architectures and compilation techniques (PACT), 2012 Sept 19–23, Minneapolis, Minnesota, USA

Top500 (2021) Top 500 supercomputer lists. [Internet]. Available from: https://www.top500.org/

Wong D, Kim NS, Annavaram M (2016) Approximating warps with intra-warp operand value similarity. In: IEEE international symposium on high performance computer architecture, March 2016, Barcelona, Spain

Power Management of Multicore Systems

17

Behnaz Ranjbar, Amit Kumar Singh, Siva Satyendra Sahoo, Piotr Dziurzanski, and Akash Kumar

Contents

Introduction . . . 562
Power Dissipation in Multicore Systems . . . 564
Causes and Effects of Power Dissipation . . . 564
Power Dissipation in Multicore Systems . . . 567
Common Power Reduction Methods . . . 568
Hardware . . . 568
Firmware . . . 569
Software . . . 571
Power Management: Embedded Systems . . . 572
Energy Minimization . . . 572
Thermal Management . . . 574
Reliability Improvement . . . 578
Power Management: Desktop and Servers . . . 580
ACPI Standard . . . 581
Power Schemes: Governors . . . 582
Power Management: High-Performance Computing (HPC) Data Centers . . . 583
Fast Heuristics . . . 584
Heuristics Using Design-Time Profiling . . . 584
Machine Learning . . . 585
Network Technologies . . . 585

B. Ranjbar · S. S. Sahoo · A. Kumar (✉)
Technische Universität Dresden, Dresden, Germany
e-mail: behnaz.ranjbar@tu-dresden.de; siva_satyendra.sahoo@tu-dresden.de; akash.kumar@tu-dresden.de

A. K. Singh
University of Essex, Colchester, UK
e-mail: a.k.singh@essex.ac.uk

P. Dziurzanski
West Pomeranian University of Technology, Szczecin, Poland
e-mail: piotr.dziurzanski@zut.edu.pl

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_55

Recent Advances in Multicore Power Management 585
2.5D/3D Systems 585
Cross-Layer Approach 586
Emerging Technologies 586
AI-/ML-Based Power Management 586
Conclusion 587
References 588

Abstract

Multicore systems have become the de facto computing platform for electronic systems, especially since 2005 when the single-core/thread performance hit the power wall. Consequently, integrating an increasing number of processing elements on a single integrated circuit has become one of the primary research goals in both architecture- and semiconductor technology-level design. However, the increasing power density in multicore systems has also led to increasing dark silicon, where a majority of the on-chip resources need to be turned off for avoiding thermal issues. To this end, intelligent power management constitutes a major focus of research in the system-level design of multicore systems. This chapter provides a brief overview of the background knowledge and the related state-of-the-art research. The chapter presents a summary of the causes and effects of power dissipation in electronic systems along with brief descriptions of the more commonly used power reduction methods. The chapter then presents the state-of-the-art research works in power management across different scales of multicore systems: embedded systems, desktops/client PCs, and HPC servers. The chapter also provides a brief overview of the more recent topics related to power management such as power dissipation in 2.5D/3D systems, cross-layer power management, and Al/ML-based power management.

Keywords

Multi/Many-core systems · Power management · AI/ML-based power management · Power-reduction mechanisms · Dark-silicon · Thermal management · Energy-minimization · Cross-layer system management

Introduction

Since around 2005, multicore processing has become the primary approach for utilizing the increasing number of transistors on a semiconductor integrated circuits (IC) (Held et al. 2006). This has also resulted in a shift of the focus in software/algorithm development to improving performance by exploiting Thread Level Parallelism (TLP). Similarly, the IC manufacturers have focused on integrating more and more cores onto the same IC to support increased TLP. This continuous effort – both in terms of software development and hardware design – has translated to an increasing portion of an IC executing at full throttle continuously. Consequently,

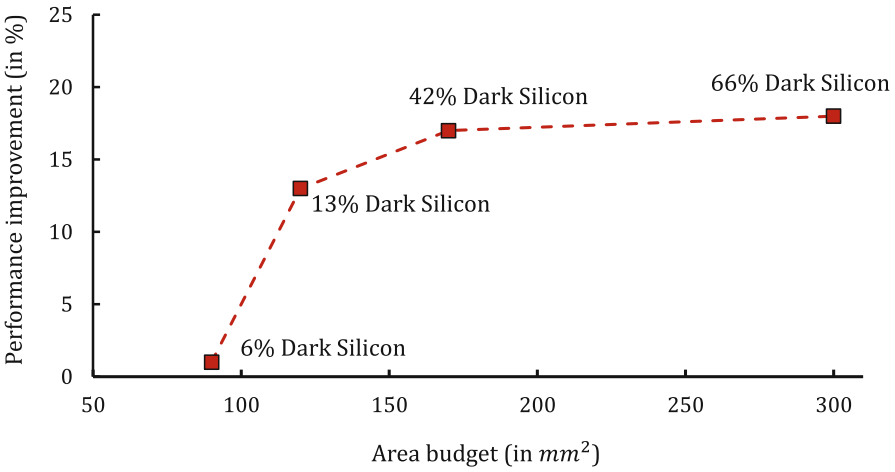


Fig. 1 Dark silicon: performance improvement with increasing area budgets and increasing proportion of dark silicon transistors. (Figure reproduced from Turakhia et al. 2013)

the very phenomenon behind the paradigm shift to multicore computing – high power density – is increasingly becoming the bottleneck to extracting higher performance from multicore systems. This phenomenon, sometimes referred to as *dark silicon* (Fig. 1), means that given the thermal and power limits of the system, with each generation of semiconductor process technology, there is a reduction in the fraction of transistors that can operate at maximum frequency (Kim et al. 2017).

Hence, power management in multicore systems involves extracting the maximum performance within the power density bounds of the system. The contributions to the increasing power density can be from multiple processing elements. In addition to actual computation, multicore processing involves varying degrees of data sharing and inter-core communication. Hence, varying levels of on-chip memory and interconnects contribute to the power dissipation, along with the computation cores. Depending upon the area of application, the multicore architecture can contain varying types and amounts of computation, communication, and memory elements. Based on the area of application, multicore systems are broadly categorized into three types: embedded systems, personal computing systems, and High Performance Computing (HPC) systems. Depending upon the type of the system, the goal of power management may vary to some extent. For instance, in some embedded systems, improving system lifetime can be equally important as reducing power and energy consumption. Similarly, the methods used for power reduction for each system may vary depending on the system's goals. These methods may involve hardware design, software design, or a combination of both. Similarly, the design decisions regarding the implementation of these methods may involve Design Space Exploration (DSE) either during compile time or runtime or a hybrid of both.

In this chapter, the aforementioned aspects of power management in multicore systems are briefly covered. Section "Power Dissipation in Multicore Systems" presents a brief background of the concepts and terms related to power dissipation that is used in the rest of the chapter. Commonly used power reduction methods are presented in section "Common Power Reduction Methods." Across sections "Power Management: Embedded Systems," "Power Management: Desktop and Servers," and "Power Management: High-Performance Computing (HPC) Data Centers", specialized power management techniques are presented for embedded, personal, and high-performance computing systems, respectively. More recent trends and advancements in power management of multicore systems are presented in section "Recent Advances in Multicore Power Management." The chapter is concluded in section "Conclusion" with a discussion on the open challenges in power management.

Power Dissipation in Multicore Systems

Power management in electronic systems is primarily targeted toward two purposes. First is to minimize heat dissipation in order to improve the system's usability (for handheld devices and wearables), reliability (for safety- and mission-critical systems), etc. Secondly, the power management methods may target the minimization of the system's energy consumption. This is crucial for battery-powered and energy-harvesting systems as well as for large-scale systems like HPC servers. The following subsections briefly cover the fundamental concepts of power dissipation – types, causes, and models – in Complementary Metal-Oxide Semiconductor (CMOS)-based circuits. Detailed analysis of these aspects can be found in Weste and Harris (2015) and Walker et al. (2019).

Causes and Effects of Power Dissipation

For any circuit element, the *instantaneous power* supplied and/or consumed is the product of the voltage and the current across the element (Eq. 1). The resulting energy supplied/consumed over a time interval T, and the average power dissipation over the same interval can be estimated as shown in Eqs. 2 and 3, respectively.

$$P(t) = I(t) \times V(t) \tag{1}$$

$$E = \int_0^T P(t)dt \tag{2}$$

$$P_{\text{avg}} = \frac{1}{T}\int_0^T P(t)dt \tag{3}$$

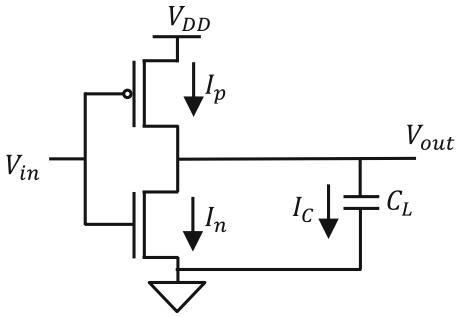


Fig. 2 Power supply and dissipation in CMOS inverter

In CMOS-based circuits, the power and energy are modeled as a function of the load driven by the circuit and the supply voltage. Figure 2 shows a CMOS inverter driving a capacitive load C_L and with a supply voltage V_{DD}. The resulting energy supplied and stored while C_L is charged through the Positive channel Metal Oxide Semiconductor (PMOS) are given by Eqs. 4 and 5, respectively.

$$E_{\text{supply}} = \int_0^\infty C_L \frac{dV}{dt} V_{DD} dt = C_L V_{DD}^2 \tag{4}$$

$$E_{\text{stored}} = \int_0^\infty C_L \frac{dV}{dt} V(t) dt = \frac{1}{2} C_L V_{DD}^2 \tag{5}$$

It can be noted that while half of the supplied energy is stored in the capacitor, the other half is dissipated as heat in the PMOS. Similarly, during discharging, the stored energy in the capacitor is dissipated in the Negative channel Metal Oxide Semiconductor (NMOS). The power dissipation during such switching of the load constitutes a major component of the total power dissipated in the system. The various power dissipation mechanisms are categorized as follows:

- **Dynamic dissipation:** This constitutes the power dissipation in the circuit as a result of the switching activity. It includes the aforementioned dissipation due to charging and discharging of the load capacitances due to switching of the gates, $P_{\text{switching}}$, and the dissipation due to the short circuit current, $P_{\text{short circuit}}$, when the pull-up and pull-down networks of the circuit are partially *ON* as the input switches. $P_{\text{switching}}$ is modelled as shown in Eq. 6, with f as the operating clock frequency. The term α, called the activity factor, represents the probability of the node switching from 0 to 1. The clock circuit has $\alpha = 1$ as the 0 to 1 transitions occur during each cycle. For other nodes in the circuit, α can vary between 0 and 0.5 and is heavily dependent on the computation task being executed.

$$P_{\text{switching}} = \alpha C_L V_{DD}^2 f \tag{6}$$

$$P_{\text{dynamic}} = P_{\text{switching}} + P_{\text{short circuit}} \tag{7}$$

- **Static dissipation:** Static power is consumed even when the nodes are not switching. This occurs primarily due to the leakage current mechanisms and any contention current in the circuit. *Subthreshold leakage* is usually the dominant contributor to the leakage current (and power) and refers to the current flowing through a transistor even when it is *OFF*. A potential difference between the source and drain terminals causes the subthreshold leakage current through the channel. Similarly, tunneling of the carriers through the dielectric between the gate terminal and the channel results in the *gate leakage* current. Other leakage mechanisms include *junction leakage*, *GIDL*, and *punchthrough* (Kim et al. 2003). Additionally, some non-CMOS circuits such as pseudo-nMOS gates, current mode logic, and many analog circuits draw currents even while quiescent. Such contention current may contribute to static power dissipation. The total power dissipation can be estimated as shown in Eq. 8.

$$P_{\text{static}} = V_{DD} \times (I_{\text{leakage}} + I_{\text{contention}}) \tag{8}$$

$$P_{\text{total}} = P_{\text{static}} + P_{\text{dynamic}} \tag{9}$$

The power dissipation mechanisms discussed above have been adversely affected by the continuous quest for higher performance through transistor scaling and architectural innovations. Increasing the operating clock frequency has been one of the primary methods of achieving faster computations. However, as shown in Eq. 6, this results in higher switching power. Consequently, scaling down of the supply voltage, along with reduced gate dimensions, had been used to maintain the power density of the ICs. However, since the failure of Dennard scaling (Dennard et al. 1974), such an approach has proved insufficient. Consequently, some power management techniques focus on managing the clock frequency based on the application's performance requirements. Similarly, since the clock tree forms the highest activity net in the design, adaptive disabling of the clock network forms an effective power management method (clock gating). Similar to dynamic dissipation, the static power dissipation has also increased considerably with technology scaling (Agarwal et al. 2004). For instance, reducing the supply voltage and the corresponding threshold voltage, V_{th}, in order to reduce power density, can lead to an exponential increase of the subthreshold leakage current. Similarly, the gate leakage current due to direct tunnelling increases exponentially with the reduction in the dielectric thickness and the increase in potential drop across the oxide. Therefore, unlike in earlier technologies (>65 nm), reducing the static power dissipation is equally important as dynamic power. Further, in systems that employ bursts of activity amid longer idle periods, managing the static power assumes higher priority. As a result, methods such as power gating and multiple power-down levels, which selectively disable power supply to multiple domains of the system, are being used extensively in power management.

The increasing power dissipation has an adverse impact on other quality metrics of the system. The higher energy consumption results in reduced usability of portable systems and increased operating costs in HPC systems. Further, the

increased power density and higher temperatures increase the demand for cooling solutions along with possible reduced performance due to thermal throttling (Bhat et al. 2019). Additionally, higher temperatures exacerbate the reliability problems of electronic systems. As a result, additional optimization objectives of *lifetime* and *functional* reliability need to be considered during the DSE for power management (Sahoo et al. 2021b).

Power Dissipation in Multicore Systems

The relationships shown in Eqs. 1, 2, 3, 4, 5, 6, 7, and 8 provide a circuit-level estimation of the power dissipation. These can be used to estimate the different power metrics at the gate- and Register Transfer Level (RTL)-level as well. Additionally, SPICE modeling along with simulations can be used to get more accurate metrics at the circuit and gate level. However, such approaches can prove to be costly for estimating power dissipation at higher abstractions such as core and system level. RTL simulations and the requirement of specific Intellectual Property (IP)s, standard cell libraries, macros, etc., for such simulations, may prove infeasible for system-level power management. Moreover, such approaches are useful during development and the related low-level design space exploration. More abstract models are required for devising and validating power management methods in multicore systems.

Measuring the power dissipation in a core for varying workloads forms a crucial step in power management. While integrating power sensors into the device seems a direct solution to this problem, it may prove too costly for many systems. Further, this method does not allow the power measurement for the DSE during development. Walker et al. (2019) provide a detailed analysis of the various approaches to power dissipation modeling for multicore systems. A *top-down* approach involves using Operating System (OS) statistics such as the core utilization to estimate power. Similarly, CPU's Performance Monitorng Counters (PMC) such as the number of integer operations and cache misses can be used to model the power dissipation. A multiple linear regression model is usually used for the top-down power estimation. A *bottom-up approach* involves using architecture simulators like *gem5* (Binkert et al. 2011) coupled with power-analysis tools such as *McPAT* (Li et al. 2013), to estimate static and dynamic power dissipation for varying workloads. While the top-down approach is suitable for runtime power management, the bottom-up approach can be used for DSE during development.

The power management-related DSE in a multicore system may use one or more of the modeling approaches discussed above. Figure 3 shows the modeling and the design abstraction levels that are involved in different DSE methodologies. *Design/compile-time* DSE usually involves making design decisions prior to the system deployment. It may include optimizing the power dissipation across various design abstraction levels – circuit, gate, RTL, microarchitecture, etc. The corresponding system-level DSE includes the scheduling of workloads across the multiple cores while satisfying the user specification. *Runtime* DSE usually involves

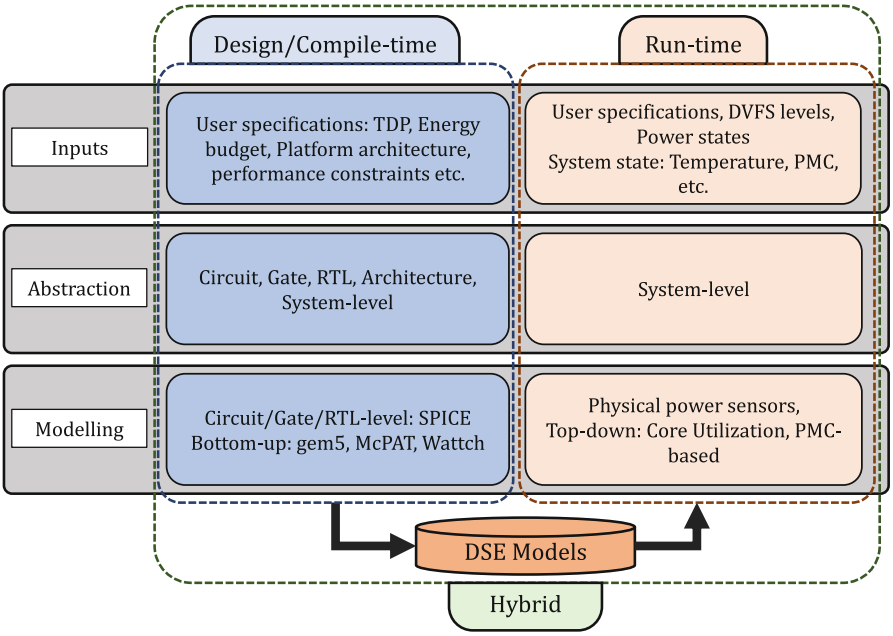


Fig. 3 Design space exploration for power management

similar scheduling, but it can utilize the runtime operating conditions, such as temperature, to enable better decision-making. However, design/compile-time DSE allows for more thorough exploration by using complex optimization methods. *Hybrid* DSE attempts to combine the best of both the approaches by deriving DSE models during design/compile time that can be used at runtime, along with dynamic operation scenario information to provide effective power management.

Common Power Reduction Methods

Hardware

The hardware resources of multicore systems include computation (core), communication (interconnects), and storage (memory) components. Figure 4 shows the contribution of each of these to the total power dissipation (Nawathe et al. 2008). Hardware-level approaches to reducing power dissipation involve architectural innovations in the design of the core, memory, and interconnect network. 3D microarchitecture is one such innovation driving the performance scaling in HPC systems. The resulting reduction in the interconnect lengths due to the vertical stacking has led to improved Performance Per Watt (PPW) compared to planar processors. However, vertical integration also leads to power density

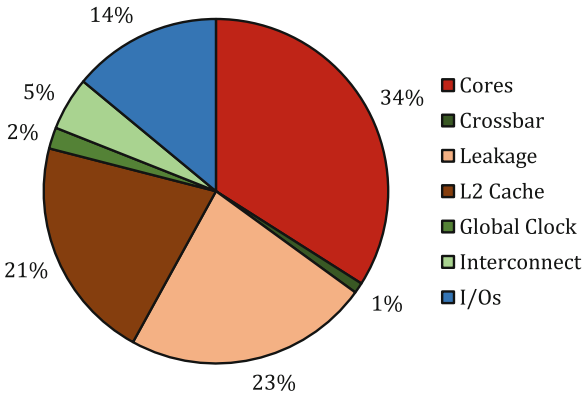


Fig. 4 Power dissipation (in % of total power) of different hardware components of Niagra2 processor @ 1.4 GHz/1.1 V (Nawathe et al. 2008)

and thermal issues due to the increased power per area. Consequently, thermal-aware 3D microarchitecture design using thermal herding techniques is being used (Puttaswamy and Loh 2007). Other core-level hardware methods involve dynamic reconfiguration of the cores to lower the power dissipation. Kontorinis et al. (2009), Rodrigues et al. (2011), and Narayanan et al. (2010) propose methods for adaptively changing the core configuration to provide the best PPW under varying workloads. Power management techniques for memory usually involve innovations in the memory hierarchy design or methods to selectively power down memory/storage components. Smart caches and drowsy caches (Flautner et al. 2002) rely on predicting cache with low leakage hardware and putting cache lines into low-power mode, respectively. Similarly, intelligent methods for putting portions of the Dynamic Random Access Memory (DRAM) into low-power mode have been proposed. A detailed survey of power management in DRAM can be found in Mittal (2012). Power management of the interconnect network usually involves an adaptive selection of routing algorithms and the reduction of the length of the interconnect wire segment (Kumar et al. 2005).

Firmware

A firmware- level technique is a software-level technique that can control the hardware. The two common types of firmware-level techniques are Dynamic Voltage and Frequency Scaling (DVFS) and Dynamic Power Management (DPM).

Dynamic Voltage and Frequency Scaling (DVFS)

DVFS method can dynamically change the voltage and/or frequency (*V-f*) of one/some/all processor cores to reduce the overall system power consumption (static and dynamic), including computation, communication, and memory parts. The power consumption, which can be varied by changing the *V-f* levels, can be written as:

$$P_{\text{total}} = P_{\text{static}} + P_{\text{dynamic}} = I_{\text{sub}} V + C_L V^2 f \tag{10}$$

where V, f, I_{sub}, and C_L are the voltage, frequency, subthreshold leakage current, and load capacitance, respectively. Here, the voltage value limits the maximum frequency. Equation 11 shows the relation between the voltage supply and frequency value (Kim et al. 2017; Pagani et al. 2018).

$$f = \beta \frac{(V - V_{th})^2}{V} \tag{11}$$

where β and V_{th} are a technology-related constant and the threshold voltage, respectively. Therefore, decreasing voltage supply leads to a decrease in operational frequency, which can vary between minimum and maximum bound (Das et al. 2013, 2014a; Ranjbar et al. 2019, 2021). As an example, employing the DVFS technique in Ranjbar et al. (2019) could reduce the energy consumption by 24% on average, compared to other recent works.

Dynamic Power Management (DPM)

DPM dynamically selects the states of the processor that can be set to active and low-power modes in general terms. The low-power modes are idle/sleep mode that can be done by the clock-gating method and off mode caused by the power-gating method.

- Clock Gating (Stop and Go): It is a method of controlling the clock signals to stop and start dynamic operations, which lead to dynamic power consumption. The clock signals are entered into computation, such as operational processors and connected memories, like registers and cache. Therefore, by stopping the clock signals and entering some (distributed policy) or all (global policy) processors into sleep mode, less dynamic power is consumed by processors and memories. When power consumption is high, this method can be applied and switched back to active mode when the thermal emergency is over. Some recent works Munawar et al. (2014) and Ranjbar et al. (2021) have used this technique to manage the power consumption by dynamically controlling the sleep cycles of the cores, which helps to keep the peak power of the chip under Thermal Design Power (TDP) and, therefore, maintain it within thermally sound operating conditions.
- Power Gating: It refers to switching off the computational processors and their connected memories and communication parts when not in use to let them decrease both static and dynamic power consumption. However, frequent on and off switches may cause more energy overheads. Therefore, this power-gating method is best used when the overall power consumption needs to be reduced significantly.

Virtualization

Virtualization is a method employed in data centers to increase utilization of the computing resources. Virtualization involves deploying multiple Virtual Machine (VMs) on the same physical server to improve the overall utilization of the resources – processor cycles or memory space – that might otherwise be lying idle if it is used by a single user. Modern server management tools such as *Microsoft System Center* can be used to recommend VMs migration allowing some physical servers to be powered down.

Software

There are some software-level techniques that can be managed by OS to reduce power consumption, like task scheduling and task migration.

Task Migration

Task migration is the runtime moving of a task/application from a hot processor core to another processor core, i.e., remap and reschedule on a colder processor core to let the hot processor core cool down. This process helps in dynamically reducing and balancing the temperature or power consumption across all processor cores in a platform (Sheikh and Pasha 2018; Henkel and Dutt 2021). This technique is mainly applied in heterogeneous multicore platforms, in which processor cores consume different power values.

Task Scheduling

Task scheduling is a process of selecting a task from an application/task set and determining where (i.e., in which core) and when to execute it (Sheikh and Pasha 2018). Choosing a processor core from a list of available processor cores helps to reduce power consumption, especially in heterogeneous multicore platforms. The task scheduling process can be static or dynamic with the aim of power reduction. In static task scheduling, the task data is known in advance. Thus, the task scheduling decision (in which processor core and the appropriate time instants to start each task's execution on the processor core) can be made at design time to reduce power consumption. In dynamic task scheduling, the start time instance of tasks and their locations are decided at runtime. Therefore, the scheduler can change the task ordering execution to reduce power consumption. In addition, the scheduler can suspend the running task during its execution or first schedule a task with a low workload if there is a high-power consumption or thermal emergency. The previous task scheduled can be resumed when the thermal crisis is over.

Data Forwarding

Data forwarding is a method used to reduce power dissipation due to frequent data accesses by the L1 cache. Modern multicore systems have a large amount of resources dedicated to reducing memory latency in the form of caches and the power

dissipation in caches can amount to up to 15% of the total power consumption. Common methods to reduce this power consumption include taking advantage of the forwarding functionality of the Load Store Queue (LSQ) to avoid data cache access. Improved methods include predicting in advance if a load matches an earlier load/store to avoid L1 cache access (Nicolaescu et al. 2003; Carazo et al. 2010).

Power Management: Embedded Systems

Energy Minimization

Energy efficiency is one of the significant metrics in modern embedded systems, such as battery-driven devices (Singh et al. 2013; Das et al. 2015b; Salehi and Ejlali 2014). Since energy is the total amount of consumed power through time, most power management techniques lead to energy minimization. All on-chip resources – computation, communication, and memory – are major contributors of energy consumption. For each resource, some of the power management techniques can be applied. Below, how the power management techniques can minimize energy in each on-chip resource is briefly described.

- *Computation:* The computational cores primarily consume high dynamic power due to executing workloads during a period. DVFS is known as a commonly effective technique for energy saving, which refers to the dynamically scaling of the voltage-frequency/voltage levels of cores. Energy is minimized in computational cores by allowing them to execute the workloads at lower *V-f* levels for a time interval. Figure 5 depicts that increasing in voltage and frequency levels leads to energy increment in each type of cores. DVFS can be applied per core(s) or task in a time period according to the system design. Besides, power gating, described earlier in section "Dynamic Power Management (DPM)," is used often in embedded systems to reduce energy.

  In heterogeneous multi-/many-core platforms, each core consumes different power values (both static power and dynamic power) (Khanh et al. 2013; Ranjbar et al. 2021). To minimize the energy consumption related to both dynamic and static power, employing task mapping and task scheduling techniques on

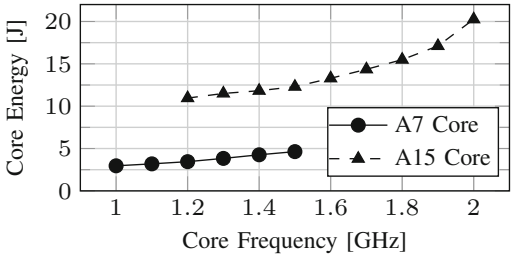


Fig. 5 Impact of varying voltage and frequency levels on energy consumption in two different types of cores (Pathania et al. 2015)

these heterogeneous cores at design/runtime is effective. As an example, three different task mapping policies in heterogeneous cores are shown in Fig. 6, in which the total energy consumption of each policy is different (Pagani et al. 2016). Heterogeneous architectures, like using Field-Programmable Gate Array (FPGA), Graphics Processing Unit (GPU), Digital Signal Processing blocks (DSPs), and CPUs, are also used in embedded systems, where the tasks' energy consumption is not the same in each architecture. Therefore, an energy-based optimization DSE is needed in these heterogeneous architectures.

- *Communication:* Multi-/many-core platforms are recently used in embedded systems to execute dependent applications. Data communication, through channels among the processors, can lead to significant energy consumption in these platforms. The communication energy is determined based on the amount of data communicated over the links and energy consumed in transmitting a single bit

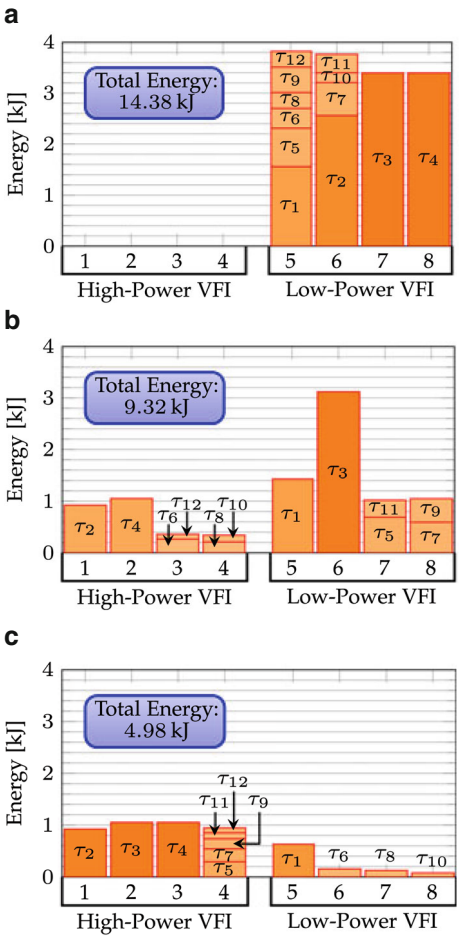


Fig. 6 Impact of different task mapping strategies on energy consumption (Pagani et al. 2016). (**a**) Task mapping policy 1. (**b**) Task mapping policy 2. (**c**) Task mapping policy 3

of data (Singh et al. 2016b; Das et al. 2013, 2014a). Therefore, selecting an application task mapping can significantly reduce task communication energy and migration overhead. In particular, using a minimum number of cores (e.g., by power gating most cores) to map the dependent tasks in the same core or neighboring cores is an approach to minimize energy consumption.

- *Memory:* Memories consume significant energy in embedded systems. Low-power techniques such as clock gating and DVFS can be applied to memories as well to reduce energy consumption (Salehi and Ejlali 2014). The leakage power of memories, such as Static Random Access Memory (SRAM), is high due to existing additional transistors for each cell (Shafique et al. 2015). Therefore, the voltage level of memories can be scaled down to reduce the leakage power and, consequently, the energy consumption of memories. Besides, clock gating can help to reduce the energy overhead of reading from memory or writing in memory. It helps by (1) managing the access and locations of bits and (2) generating appropriate signals to adapt the data at reading and writing memory ports (Shafique et al. 2015).

  In different types of memories, ScratchPad Memory (SPM)s are more energy-efficient than SRAMs and caches. Since using a low-energy memory is desirable for embedded systems, SPM is used as on-chip memory. Although its static power consumption is high, the read and write time access is very low, compared to other memories like SRAMs, which leads to low-energy consumption (Shekarisaz et al. 2021).

Thermal Management

Most embedded systems are battery-operated in which managing the energy consumption is critical. Hence, since leakage current increases exponentially with temperature elevating, which generates a high level of energy consumption, thermal management is becoming one of the main design objectives (Cox et al. 2013). Moreover, it assists in increasing lifetime reliability by decreasing power density (Das et al. 2014d). Thermal management in embedded systems can be fulfilled by reducing the average temperature and/or maximum temperature. The temperature of each processor core depends on its power consumption and temperature of its neighbors (Chantem et al. 2010). Hence, managing the power consumption of processor cores helps to control the temperature in these systems. According to Moore's Law, an exponential increase in power density of processor cores leads to thermal issues. Although cooling components, such as heat sink and fan, are significant components in modern-embedded platforms, they are very costly and bulky (Chantem et al. 2010) and may not be appropriate for most embedded systems, like portable systems. Consequently, multiple techniques of power management, discussed in section "Common Power Reduction Methods," can be used in each category, which is explained in detail as follows.

- *Maximum Temperature Reduction*: Increasing the maximum temperature may cause the hotspot in the hardware platforms, which leads to unreliability, or

may stop or restart the system by an integrated Dynamic Thermal Management (DTM) unit (Al Faruque et al. 2010; Ranjbar et al. 2021). Therefore, some power management techniques can be used to reduce the maximum temperature and avoid hotspots.

As mentioned in section "Dynamic Voltage and Frequency Scaling (DVFS)," DVFS/Dynamic Voltage Scaling (DVS)/ Dynamic Frequncy Scaling (DFS) are exploited for power density reduction. Figure 7 presents the variation of a core's temperature under different core's voltage levels (0.8 to 1.2. V) for four different activity scenarios of its neighbor cores ($Core_e$, $Core_w$, $Core_n$, $Core_s$). This figure shows that the core's temperature is less at lower voltage levels in each combination of neighbor cores' activity. Analyzing and controlling the system's temperature by scaling down the voltage or/and frequency can be done in both design-time and runtime phases. To avoid hotspots and reduce the chip's maximum temperature, this technique can be applied to a processor core or a cluster (a set of processor cores), which has a higher power density than other processor cores/clusters. Finding the optimum and best voltage/frequency

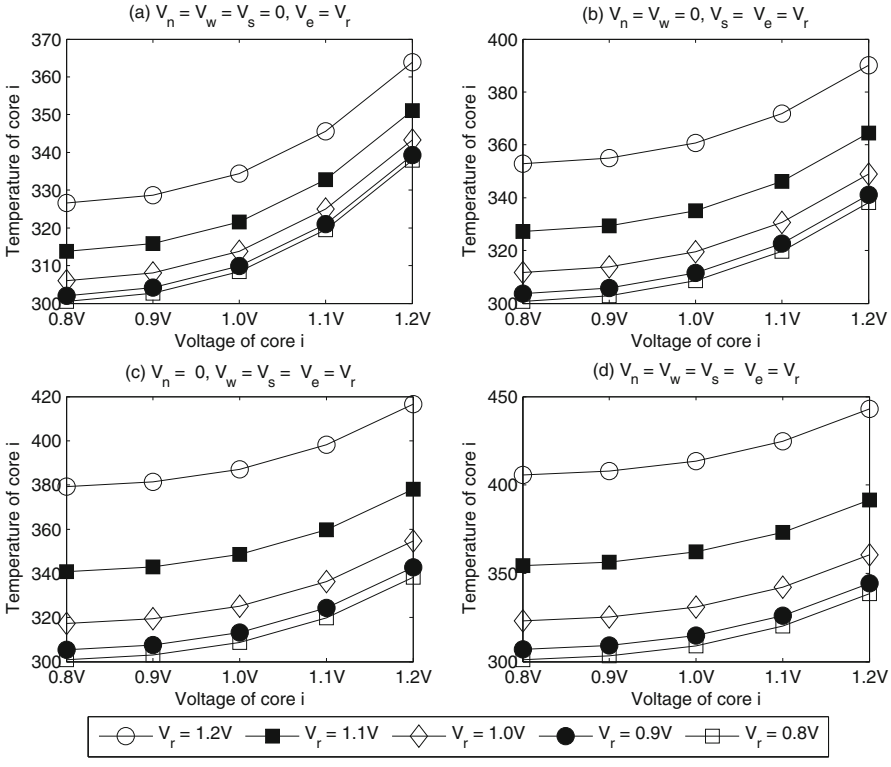


Fig. 7 Impact of varying voltage level on cores' temperatures (in Kelvin) (Das et al. 2014b)

level can be done by various methods, such as proposing heuristics, algorithms, or learning techniques while using the power density value of processor cores or reading temperature values from sensors. Reducing the maximum temperature can be controlled by defining a temperature constraint for processor cores or keeping the maximum temperature at the least possible value. However, reducing the supply voltage may exacerbate the noise margin due to reducing the gap between the supply voltage and the threshold voltage. Therefore, leveraging a full voltage swing is another technique that can be used to manage the maximum temperature.

Stop-go and power gating can also help cool down the hot processor cores and reduce the maximum temperature. These techniques are applied until the maximum temperature is reduced up to the optimum value, and then the processor cores can resume their execution. In order to have a safe operation, stop-go task scheduling algorithms have been proposed to minimize the maximum temperature optimally. Besides, these techniques can be applied for task (re-)mapping in multicore platforms to manage the hotspots. Indeed, when the temperature of a processor core is increasing more than the thermal threshold, the hot processor core can switch from the busy mode to idle/sleep modes and (re-)map/migrate the dynamic operations (i.e., computation of running tasks) to cold processor cores. Furthermore, from the perspective of thermal management in many-core platforms, a TDP budget is defined not to activate all processor cores of the chip simultaneously. TDP is the maximum sustainable power that a chip can dissipate within the safe range, i.e., below the defined maximum temperature (Ranjbar et al. 2022). Thus, some processor cores are power-gated, which leads to power density reduction and, consequently, avoiding the hotspots. The impact of considering a TDP constraint while mapping tasks on cores can be observed in an example of Fig. 8. Figure 8a presents the power traces of two different methods of task mapping, where power gating is used in the method of Ranjbar et al. (2022), to manage the TDP constraint. As illustrated in Figs. 8b,c, since the maximum temperature is managed by mapping the tasks on cores, the maximum temperature achieved using the method of Ranjbar et al. (2022) is less than the other one.

- *Average Temperature Reduction*: DVFS, stop-go, and power gating are also popular techniques that are employed similarly to manage the average temperature. A system power and temperature traces of two cores can be seen in Fig. 9, in which the runtime DVFS technique is applied to two cores within a cluster. In this figure, a DVFS-based method is proposed to decrease the average core temperature. Based on the proposed task scheduling, the DVFS technique is applied from the middle to the end of the period (shown in Fig. 9a), and consequently, the temperature of cores is decreased. In addition to these mentioned techniques, clock gating is a technique that can be used to reduce the total power consumption and, consequently, the overall temperature. Indeed, in the case of thermal issues, the proper clock throttling ratio is determined to drop the temperature sharply. In addition to reducing the average temperature by reducing the dynamic power, leakage power reduction can help by using

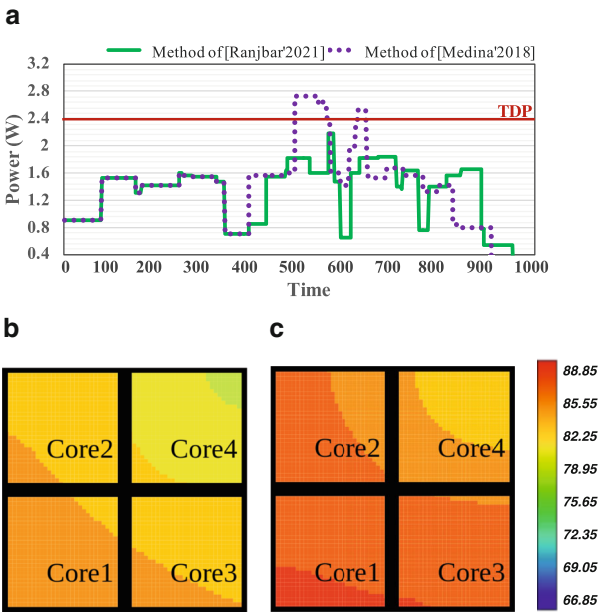


Fig. 8 Impact of considering a TDP constraint on system's maximum temperature through the use of power-gating technique. (**a**) Power traces for two methods of Ranjbar et al. (2022) and Medina et al. (2018). (**b**) Temp. profile of Ranjbar et al. (2022) method. (**c**) Temp. profile of Medina et al. (2018) method

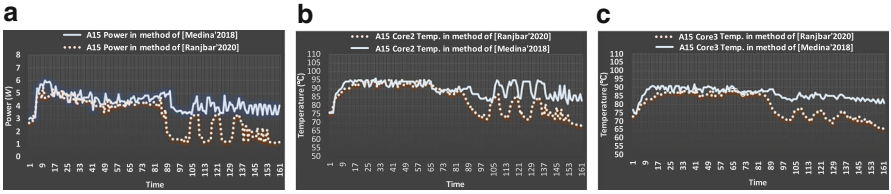


Fig. 9 The relation between cores' temperature and power consumption for method of Medina et al. (2018) and method of Ranjbar et al. (2021) by using DVFS technique. (**a**) Power trace of the big cluster. (**b**) Temperature trace of A15-core2. (**c**) Temperature trace of A15-core3

compiler-directed thermal management techniques, such as Instruction Per Cycle (IPC) tuning. Indeed, in the case of defining fixed active numbers in a loop (e.g., using four out of six Algorithm Logic Unit (ALU)s in a loop), IPC tuning turns off unused functional units (ALUs in this example) to reduce the leakage power. In addition, the cache also plays an important role in causing thermal issues. As a result, controlling the cache access by selecting the cache ways reduces the power density and, consequently, reduces the average temperature (Amrouch et al. 2013).

Reliability Improvement

Reliability is a major concern for most safety-critical applications executed on embedded systems. In these applications, both functional reliability and lifetime reliability constraints are essential design considerations and must be guaranteed (Sahoo et al. 2021b). Functional reliability represents the probability of correctness of application results, and lifetime reliability indicates the system's operational life (Sahoo et al. 2018, 2021b). Various techniques, like low-power techniques, lead to reliability improvements. Below, the power management techniques used to improve lifetime reliability are first explained and then the techniques used for functional reliability improvement are explained.

- *Lifetime Reliability:* In battery-based embedded systems, like most mobile computing devices, the power is a hard constraint on their operational lifetime. A system's lifetime depends on multiple wear-out effects, such as Electromigration (EM), Negative Bias Temperature Instability (NBTI), Time Dependent Dielectric Breakdown (TDDB), and Thermal Cycling (TC). Most of these wear-out effects are influenced by operating temperature (Das et al. 2014b,d, 2018; Navardi et al. 2022). Therefore, improving the temperature using low-power techniques, such as DVFS, power gating, clock gating, task mapping/migration, and scheduling, leads to lifetime reliability improvement and longevity insurance. Figure 10 is an example of using the task (re-)mapping technique that leads to different values of lifetime reliability for three cores through the time (Das et al. 2015a). Core2 is stressed more than the other two cores and ages at time τ_0. Thus, the tasks are remapped from Core2 to other cores to improve the lifetime reliability. In

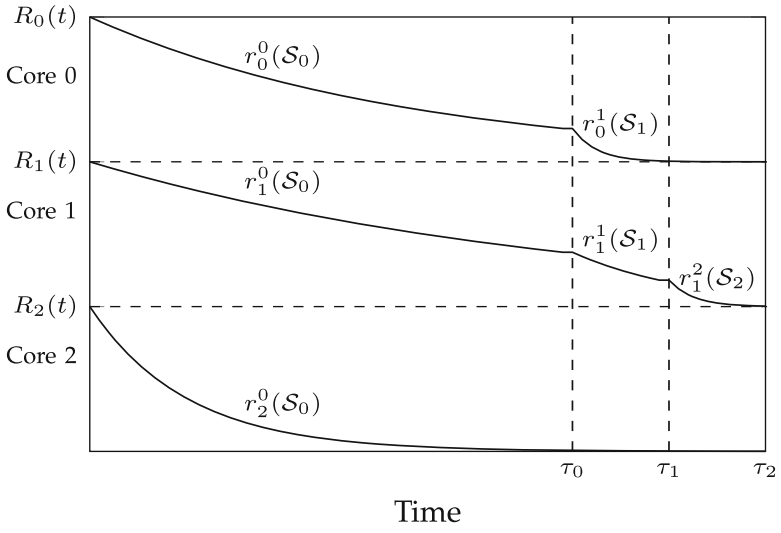


Fig. 10 Impact of task remapping on lifetime reliability (Das et al. 2015a)

addition, NBTI can degrade the lifetime of caches as well. Balancing the aging of memory cells in terms of energy consumption leads to a longer lifetime (Shafique et al. 2015). Besides, reducing the leakage power by controlling the threshold voltage is a technique that can manage the NBTI (Gnad et al. 2015).

- *Functional Reliability:* The output correctness of tasks should be investigated on the whole chip, including computations, memories, and communications. Soft Error Rate (SER) management is an approach to managing the functional reliability of computation parts. SER is the probability of soft errors occurring during the time interval, which depends on the operating frequency (Ma et al. 2018, 2019). It indicates that improving core frequency is effective in enhancing SER. Therefore, DVFS is one of the low-power techniques to manage functional reliability. Munawar et al. (2014) explores the impact of *V-f* scaling on the functional and thermal reliability. It shows that the fault rate is increased in low frequencies, which causes a degradation in functional reliability (Sahoo et al. 2021a). Figure 11 depicts the impact of applying different frequency levels on fault rate and functional reliability for an application. The fault rate is computed based on Eq. 12, which is mentioned in Das et al. (2014c). Here, λ_0, d, f, and f_{min} are soft error rate at the maximum frequency level, architecture-specific constant, frequency level, and the minimum frequency level, respectively. However, thermal reliability (i.e., the lifetime reliability that is influenced by temperature variation) is decreased at high-frequency levels, which is in contrast to functional reliability optimization. The thermal reliability at time instance t due to the EM is given by following equation, where n, E, E_a, β, and k are material-based constant, energy consumption, activation energy, Weibull sloop parameter, and Boltzman constant, respectively (Dinakarrao et al. 2019). The power consumption and temperature would be higher at high frequencies, which leads to a decrease in thermal reliability according to Eq. 13. Therefore, selecting the optimum levels of voltage and frequency is crucial. In addition,

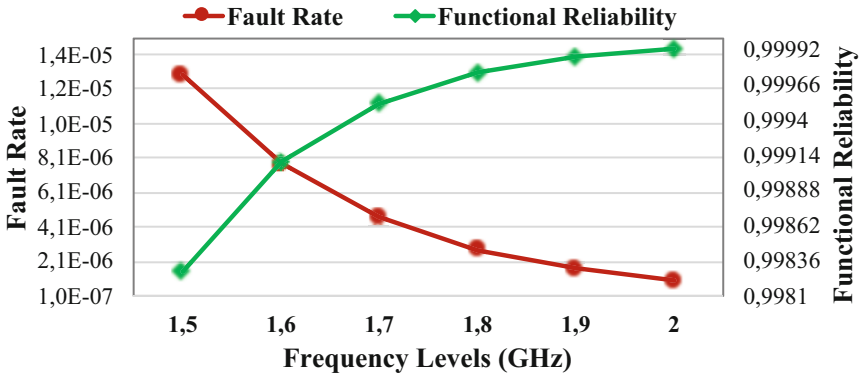


Fig. 11 Impact of varying frequency levels on fault rates and functional reliability

power gating can help to improve SER by reducing the total utilization of cores and power gating them.

$$\lambda(f) = \lambda_0 10^{\frac{d(1-f)}{1-f_{min}}} \tag{12}$$

$$R^{thermal}(t) = e^{-C \times t^{\beta} \times e^{\frac{E_a \beta}{k \times Temp}}}, \quad C = \left(\frac{1}{\Gamma \times (1 + \frac{1}{\beta}) \times E^{-n}}\right)^{\beta} \tag{13}$$

Increasing power consumption may lead to a rise in the Bit Error Rate (BER) of communication parts and, consequently, unreliable packet transmission. Therefore, considering a power budget and selecting the optimum route for data transfer improves the BER and reliable packet transmission (Brahim and Khan 2006). In memories, consuming low leakage power leads to reliable read and write operations. Most of the SRAM-based architectures while using Non Volatile Memory (NVM) have high leakage power consumption and low reliability. Therefore, proposing a nonvolatile SRAM storage element (i.e., a memory architecture) with low leakage power can reduce the reliability issues. To manage the leakage power, if Phase Change Memory (PCM) (one of the most promising resistive NVMs) is used in memory architectures, power gating can be used in PCM cells when they do not have any active leakage power (Huang et al. 2014).

Power Management: Desktop and Servers

A typical usage pattern of a desktop computer is interactive and user-centric, where a computer performs some action based on user commands. Typically, the load of a desktop computer is diverse but characterized by an overrepresentation of idle periods. Such imbalance of processor business influences the power utilization of the entire computation to a large degree. For example, Mahesri and Vardhan (2005) determined this difference experimentally, measuring the power usage based on data from an oscilloscope and current probes. According to these measurements, the entire computer system consumed 8.23 to 25.8 Watts depending on the system idleness, the backlight level, and the DVFS presence. That paper also presented a detailed breakdown of a laptop energy consumption concerning the key components, namely, CPU, hard drive, wireless card, LCD, backlight, optical drive, memory system, and graphics card. Some results from that breakdown are shown in Fig. 12. From this figure, it follows that CPUs can dissipate more than 50% of the total system power only for CPU-intensive tasks. This value is reduced up to 5% for an idle machine.

A typical server workload and the resulting power consumption is also prone to change significantly in response to users' activity. An example of CPU usage and power dissipation for HTTP web server, database server, and computing server is

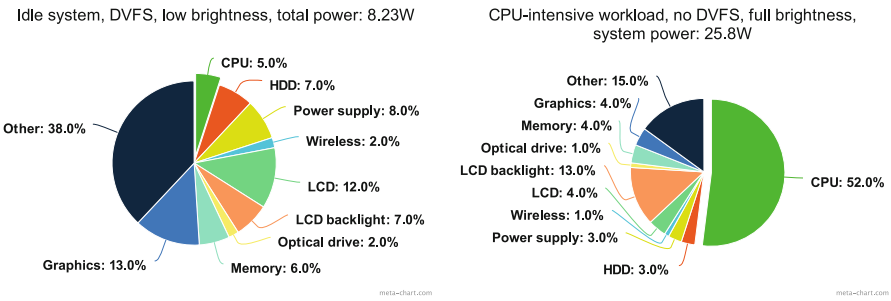


Fig. 12 Breakdown of system power conversion with various workload, CPU speeds, and display brightness levels (Mahesri and Vardhan 2005)

provided in Aroca and Gonçalves (2012) for a growing number of simultaneous clients and computer architecture. That experiment also showed a significant increase in response time with the growth of the number of clients. Hence, it may be concluded that a single server machine can be treated as similarly to desktop computers and hence, the techniques enumerated in this section apply. However, to provide a satisfying quality of service regarding the response time regardless of the surge of the number of simultaneous clients, dynamic scaling in and out of the number of servers needs to be provided. Such service scaling is available in HPC Data Centers, which are targeted in section "Power Management: High-Performance Computing (HPC) Data Centers."

ACPI Standard

From the above-referred experiment, it follows that energy efficiency in the case of user-centric desktop computers requires DPM and DVFS techniques described earlier in subsections "Dynamic Power Management (DPM)" and "Dynamic Voltage and Frequency Scaling (DVFS)." Consequently, modern CPUs used in desktop and server computers follow the OS-based Advanced Configuration and Power Interface (ACPI) standard (https://uefi.org/). This standard defines mechanisms for putting the desktop computer as a whole in and out of system sleep states. However, regarding the subject of this chapter, the most crucial is the ACPI section entitled "Processor Configuration and Control", which describes the configuration and control of the processor's power and performance states.

In ACPI, C0 is defined as the operating state of a processor, whereas C1, C2, and C3 are the halt, stop-clock, and sleep states, respectively. Additional C-states have been introduced by several chip vendors, such as C6 in Intel Xeon, which is known as deep power conservation state. The Haswell processors can enter C7, a deep power down state, whereas even lower power dissipation is achievable in the deeper power down states labelled as C8/C9/C10 in the 8th and 9th generation of Intel Core processor families and Intel Xeon E processor families.

While running in the operating state C0, a core can be in one of the predetermined power-performance states known as P-states that select the DVFS level. Reduced performance and lower energy dissipation are characteristics of P-states with higher indices. Various parameters supplied by monitoring infrastructure tools and services can be used to make an informed decision on voltage scalings, such as infrastructure utilization or latency between the input and output timestamps.

Power Schemes: Governors

In the presence of the DVFS facilities, selecting a core for a task becomes a more difficult problem even for multicores with homogeneous architecture since their cores can function at a varied voltage and frequency levels at the same time instant. Such scheduling policies considering various voltage and frequency levels are referred to as *voltage scheduling*.

In a multicore CPU, a task can be assigned to a core statically or dynamically. The former is recommended when the workload is known ahead of time, whereas dynamic allocation, which occurs after a task is released, is the only option for workloads that are not known in advance. Dynamic task mapping on CPUs that enables DVFS is even more difficult, because not only must the target core be selected but also the level of accessible voltage. The scheduling methods in Windows or Linux (from kernel version 2.6) that use dynamic frequency scaling for contemporary multicore processors from Intel (SpeedStep technology) and AMD (PowerNow! or Cool'n'Quiet technology) are good examples of this type of allocation.

In contemporary operating systems used in desktops and servers, the available power schemes for CPUs are termed *governors*. In Linux, the *performance* governor runs the CPU at the maximum frequency, whereas the *powersave* governor activates the minimum frequency. In the case of a high load, the *ondemand* governor selects P0 immediately, whereas the conservative governor progressively modifies the current P-state. The purpose of these latest two heuristics is to keep CPU utilization near 90% by reactively decreasing or raising frequency with particular heuristics. The governor proposed in Ayoub et al. (2011), which after being implemented in Linux Kernel and tested on a 32 nm Intel hexa-core dual-socket Westmere Xeon, managed to reduce the standard deviation from target performance by more than 90% over the state-of-the-art policies while reducing average power by 17% while applied for Spec2K and SpecWeb benchmarks.

Per-core or per-chip (chip-wide) custom governors are available. In Kim et al. (2008), an interesting comparison of per-chip and per-core DVFS is shown, according to which per-core DVFS saves roughly 20% more energy than a standard chip-wide DVFS using off-chip regulators. Despite such improvement potential, per-core DVFS has not been implemented widely in hardware. For example, the active cores even in the third generation of Intel x86 processors (Ivy Bridge) had to work with the same frequency and voltage in steady states, whereas their competing cores in AMD processors could operate with various frequencies, but still with a

single voltage value, required by the core in the fastest P-state. The fourth Intel x86 generation CPUs (Haswell) were the first to offer per-core DVFS capability in production. This support was removed from later generations Sky Lake and Kaby Lake. The authors of Acun et al. (2019) adopted a fine-grained function-level per-core DVFS technique to construct an intelligent energy-efficient runtime module. Over the initial iterations, that module discovered the energy-optimal frequency for each function of the analyzed application and then used that optimal frequency in subsequent iterations. On Haswell CPUs, they achieved a 4 percent to 35 percent energy reduction over chip-level DVFS while maintaining performance (Acun et al. 2019).

In Zhu et al. (2020), authors explored the relationship between per-core DVFS and phase scaling of the voltage regulator (VR) to achieve system-level energy optimization. The proposed convex-optimization model has been split into offline and online stages, which reduced the optimization time without incurring energy overhead. Their proposed model has been tested on platforms with four, eight, and sixteen cores where it lowered the system energy usage by up to 22.4 percent with good scalability on the testing data.

Power Management: High-Performance Computing (HPC) Data Centers

An HPC data center connects a set of nodes (servers) (Singh et al. 2015b), where each node contains a set of cores within a chip and the cores communicate via an interconnection network and the nodes communicate via a high-speed network, for example, InfiniBand. The size and performance of these systems continue to increase but at the expense of high-energy consumption. Therefore, it is important to take measures that can help reduce the energy consumption.

The energy is consumed to execute the queued jobs that arrive periodically or randomly. The data center companies make profit by executing the jobs that are associated with some values. Value is typically assigned based on expected profit earned from the completion of the job. In data centers, typically, the scheduling of jobs is influenced by their value, i.e., a resource management tries to maximize the profits by allocating the limited resources to the highest-value jobs in the queue. However, with the rise of power/energy consumed by data centers, the resource management needs to take energy management into account.

The literature has advanced with techniques to improve the energy efficiency of the data center. The popular techniques are based on mainly Dynamic Power Management (DPM) and Dynamic Voltage Frequency Scaling (DVFS) principles. Recently, virtual machines (VMs) consolidation that focuses on reducing the number of servers to run VMs has become a major focus area (Sun et al. 2015).

To achieve energy efficiency and/or high profit, existing approaches apply a variety of techniques, for example, (1) fast heuristics to quickly find practical allocations for the dynamically arriving jobs as they reduce the delay in allocations, (2) design-time profiling to reduce runtime computational complexity (Singh et al.

2015b), (3) machine learning, and (4) novel data center network (DCN) technologies, leveraging emerging interconnection paradigms such as millimeter-wave (mmWave) interconnects, Ethernet and wireless. The details of these techniques are as follows.

Fast Heuristics

Market-inspired heuristics have shown promising results in the common situations where the HPC data center is overloaded with more jobs than they can handle at the same time. The jobs are typically placed in a job queue and picked from it for allocation based on various principles.

Several heuristics choose the highest-value job first. However, it might end up consuming many resources and thus leaving limited resources for jobs arriving in the future. To overcome such problems, some heuristics choose a job based on its value density, which is typically computed as the ratio of value over resources requirement or with some other variants (Bansal and Pruhs 2010). There has also been effort to preempt execution of low-value jobs and assign freed resources to high-value arrived jobs (Singh et al. 2015a). However, many such heuristics do not consider optimization of energy consumption.

As mentioned earlier, the heuristics considering energy consumption make use of VMs consolidation and DVFS. The consolidation process places VMs with low utilization to a single server so that other used servers can be freed to shut them down. DVFS approaches are also well studied (Singh et al. 2015b). Energy-aware resource allocation has also been used to save energy consumption (Jin et al. 2019). However, most of the approaches considering energy optimization ignore profit optimization. The approaches optimizing profit and energy are the focus of several recent research works (Singh et al. 2015b; Mamun et al. 2020).

Heuristics Using Design-Time Profiling

The use of design-time profiling results in the runtime resource management process has resulted in better joint optimization of both the profit and energy as heavy computations are shifted to design time (Singh et al. 2015b). These approaches have helped to make fast decisions at runtime and thus in developing fast runtime heuristics, mentioned in the previous subsection. Further, design-time profiling has helped in adapting the allocations during tasks' execution (Singh et al. 2016a).

Although such design-time profiling has shown promising results, the technique is not flexible to all the varying situations as it performs profiling for a set of situations and assume that there is little deviation in resources required by the jobs in the system. To overcome these disadvantages, algorithms that account for varying situations can be designed. It can be done by taking a full history of information about jobs in real time. This has led to the development of various machine learning approaches.

Machine Learning

Reinforcement learning technique for scheduling tasks on large-scale distributed systems and data centers (Lin et al. 2016) have been explored. However, these techniques consider the fulfillment of the service-level agreement (SLA) that provides a fixed value when jobs are completed before a specified deadline. In Mamun et al. (2020), the technique considers more generalized case where the value of a job changes gradually as a function of time, known as a value curve.

Deep reinforcement learning to control various optimization aspects has also been explored, for example, resource allocation and power management for energy optimization (Liu et al. 2017) and resource provisioning and task scheduling to reduce energy consumption (Oudaa et al. 2021). Such system-level optimizations are expected to rise in the future.

Network Technologies

Since network consumes significant amount of energy, novel data center network (DCN) technologies have been explored to reduce energy consumption. Millimeter-wave (mmWave) interconnects have been proposed to reduce the power consumption of the networking equipment. Most of these works are for wireless data centers and propose interconnecting entire racks of servers as units with 60 GHz wireless links. Alternatively, server-centric wireless DCNs where direct wireless links are used for server-to-server communication have also been explored (Mamun 2021). These wireless data center architectures are promising alternative to traditional wired architecture for HPC computing for achieving further reduction in power consumption.

Recent Advances in Multicore Power Management

2.5D/3D Systems

In order to address the limitations of two-dimensional (2D) multicore systems like signal transmission delay and large chip area, research has progressed to explore to design 2.5D (PD et al. 2017) and 3D (Li et al. 2019) systems. However, designing such systems has additional challenges, mainly high power density leading to higher operating temperatures, which leads to an unreliable system and degraded performance. This demands for thermal-aware integration of components in both 2.5D and 3D systems. Additionally, thermal-aware resource management strategies have been developed to fulfill the demand for high-performance computations while keeping the system reliable. The development of novel integration and resource management strategies are expected to continue to cater to the needs of emerging computation and communication-intensive applications.

Cross-Layer Approach

Traditional DSE for power management – both during compile time and runtime – involved optimization of the methods for a single abstraction layer. In contrast, a cross-layer approach involves joint optimization across multiple layers. For instance, Sahoo and Kumar (2021a) presents optimization of the choices for DVFS and multiple implementations of a task, with varying activity factors. However, the joint exploration also results in the explosion of the related design space. As a result, novel approaches to the search for the optimal design points are being adopted for cross-layer DSE for power and other quality metrics (Carter et al. 2010; Sahoo et al. 2016). These include constrained decoding (Sahoo and Kumar 2021a), Multi-Objective Evolutionary Algorithms (MOEA) (Sahoo et al. 2019, 2020) and Monte Carlo Tree Search (MCTS) (Sahoo and Kumar 2021b), among others.

Emerging Technologies

The last decade has seen rapid proliferation of electronic systems across application areas, powered by various emerging technologies – both in devices and applications. The emerging applications technologies such as *edge-AI*, Internet of Thing (IoT), etc., have been mainly driven by the rapid strides in the field of AI/ML and have their own unique set of power-management challenges such as battery-less edge devices, computation/communication trade-offs for IoT, etc. Further, Artificial Intelligence (AI) algorithms such as Deep Neural Networks (DNN) require simpler computations and far more memory accesses compared to more traditional computation tasks. Hence, the power management of *noncore* and *uncore* components has become equally critical. The emerging device technologies are driven by the need to reduce leakage power at shrinking transistor dimensions.

AI-/ML-Based Power Management

It is evident that artificial intelligence (AI)/machine learning (ML) has penetrated almost all the fields of computation and power management of multicore systems is no different (Singh et al. 2020). Due to the dynamic nature of application workloads on multicore systems, reinforcement learning (RL) has been extensively explored (Dey et al. 2020). Model-based learning has also been a popular choice as they enable prediction of the current and future states of a system (Isuwa et al. 2022). These recent activities are expected to advance in the future in various forms like distributed RL for large-scale multicore systems to address scalability issues, multi-objective optimizations (e.g., performance, power, and security) that are typically not efficient and extending AI/ML strategies to design and optimize systems for various complex application domains.

Conclusion

With the perpetual quest for high performance and the resulting technology, scaling power management has become one of the primary system-level design objectives. The recent proliferation of AI/ML into every sphere of our lives, and the need for high-performance systems to execute the related applications, multicore systems will need to scale across multiple dimensions – number of cores, heterogeneity of cores, microarchitecture, packaging, etc. Implementing both traditional and emerging applications in multicore systems varying across these aspects provides ample scope for innovations in power management. To this end, this chapter provides an overview of the fundamental aspects, the state of the art, and a peak into the future of power management in multicore systems.

Glossary

AI Artificial Intelligence.
ALU Algorithm Logic Unit.
BER Bit Error Rate.
CMOS Complementary Metal-Oxide Semiconductor.
DFS Dynamic Frequncy Scaling.
DNN Deep Neural Networks.
DPM Dynamic Power Management.
DRAM Dynamic Random Access Memory.
DSE Design Space Exploration.
DSPs Digital Signal Processing blocks.
DTM Dynamic Thermal Management.
DVFS Dynamic Voltage and Frequency Scaling.
DVS Dynamic Voltage Scaling.
EM Electromigration.
FPGA Field-Programmable Gate Array.
GPU Graphics Processing Unit.
HPC High Performance Computing.
IC integrated circuits.
IoT Internet of Thing.
IP Intellectual Property.
IPC Instruction Per Cycle.
LSQ Load Store Queue.
MCTS Monte Carlo Tree Search.
MOEA Multi-Objective Evolutionary Algorithms.
NBTI Negative Bias Temperature Instability.
NMOS Negative channel Metal Oxide Semiconductor.
NVM Non Volatile Memory.

OS Operating System.
PCM Phase Change Memory.
PMC Performance Monitorng Counters.
PMOS Positive channel Metal Oxide Semiconductor.
PPW Performance Per Watt.
RTL Register Transfer Level.
SER Soft Error Rate.
SPM ScratchPad Memory.
SRAM Static Random Access Memory.
TC Thermal Cycling.
TDDB Time Dependent Dielectric Breakdown.
TDP Thermal Design Power.
TLP Thread Level Parallelism.
VMs Virtual Machine.

References

Acun B, Chandrasekar K, Kale LV (2019) Fine-grained energy efficiency using per-core DVFS with an adaptive runtime system. In: 2019 Tenth International Green and Sustainable Computing Conference (IGSC), pp 1–8. https://doi.org/10.1109/IGSC48788.2019.8957174

Agarwal A, Kim CH, Mukhopadhyay S, Roy K (2004) Leakage in nano-scale technologies: mechanisms, impact and design considerations. In: Proceedings of the 41st Annual Design Automation Conference, DAC'04. Association for Computing Machinery, New York, pp 6–11. https://doi.org/10.1145/996566.996571

Al Faruque M, Jahn J, Ebi T, Henkel J (2010) Runtime thermal management using software agents for multi-and many-core architectures. IEEE Des Test Comput 27(6):58–68

Amrouch H, Ebi T, Schneider J, Parameswaran S, Henkel J (2013) Analyzing the thermal hotspots in fpga-based embedded systems. In: 2013 23rd International Conference on Field programmable Logic and Applications. IEEE, pp 1–4

Aroca RV, Gonçalves LMG (2012) Towards green data centers: a comparison of x86 and ARM architectures power efficiency. J Parallel Distrib Comput 72(12):1770–1780. https://doi.org/10.1016/j.jpdc.2012.08.005, https://www.sciencedirect.com/science/article/pii/S0743731512002122

Ayoub RZ, Ogras U, Gorbatov E, Jin Y, Kam T, Diefenbaugh P, Rosing T (2011) OS-level power minimization under tight performance constraints in general purpose systems. In: Proceedings of the 17th IEEE/ACM International Symposium on Low-Power Electronics and Design, ISLPED '11. IEEE Press, Fukuoka, Japan, pp 321–326. https://doi.org/10.1109/ISLPED.2011.5993657

Bansal N, Pruhs KR (2010) Server scheduling to balance priorities, fairness, and average quality of service. SIAM J Comput 39(7):3311–3335

Bhat G, Gumussoy S, Ogras UY (2019) Power and thermal analysis of commercial mobile platforms: experiments and case studies. In: 2019 Design, Automation Test in Europe Conference Exhibition (DATE), pp 144–149. https://doi.org/10.23919/DATE.2019.8714831

Binkert N, Beckmann B, Black G, Reinhardt SK, Saidi A, Basu A, Hestness J, Hower DR, Krishna T, Sardashti S, Sen R, Sewell K, Shoaib M, Vaish N, Hill MD, Wood DA (2011) The gem5 simulator. SIGARCH Comput Archit News 39(2):1–7. https://doi.org/10.1145/2024716.2024718

Brahim GB, Khan B (2006) Budgeting power: packet duplication and bit error rate reduction in wireless ad-hoc networks. In: Proceedings of the 2006 International Conference on Wireless Communications and Mobile Computing, pp 293–298

Carazo P, Apolloni R, Castro F, Chaver D, Pinuel L, Tirado F (2010) L1 data cache power reduction using a forwarding predictor. In: Proceedings of the 20th International Conference on Integrated Circuit and System Design: Power and Timing Modeling, Optimization and Simulation, PATMOS'10. Springer, Berlin/Heidelberg, pp 116–125

Carter NP, Naeimi H, Gardner DS (2010) Design techniques for cross-layer resilience. In: 2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010), pp 1023–1028. https://doi.org/10.1109/DATE.2010.5456960

Chantem T, Hu XS, Dick RP (2010) Temperature-aware scheduling and assignment for hard real-time applications on MPSoCs. IEEE Trans Very Large Scale Integr (VLSI) Syst 19(10):1884–1897

Cox M, Singh AK, Kumar A, Corporaal H (2013) Thermal-aware mapping of streaming applications on 3D multi-processor systems. In: Proceedings of IEEE Symposium on Embedded Systems for Real-time Multimedia, pp 11–20. https://doi.org/10.1109/ESTIMedia.2013.6704498

Das A, Kumar A, Veeravalli B (2013) Communication and migration energy aware design space exploration for multicore systems with intermittent faults. In: 2013 Design, Automation Test in Europe Conference Exhibition (DATE), pp 1631–1636. https://doi.org/10.7873/DATE.2013.331

Das A, Kumar A, Veeravalli B (2014a) Energy-aware task mapping and scheduling for reliable embedded computing systems. ACM Trans Embed Comput Syst (TECS) 13(2s):1–27

Das A, Kumar A, Veeravalli B (2014b) Temperature aware energy-reliability trade-offs for mapping of throughput-constrained applications on multimedia MPSoCs. In: 2014 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, pp 1–6

Das A, Kumar A, Veeravalli B, Bolchini C, Miele A (2014c) Combined DVFS and mapping exploration for lifetime and soft-error susceptibility improvement in MPSoCs. In: Proceedings on Design, Automation & Test in Europe Conference & Exhibition (DATE), pp 1–6. https://doi.org/10.7873/DATE.2014.074

Das A, Shafik RA, Merrett GV, Al-Hashimi BM, Kumar A, Veeravalli B (2014d) Reinforcement learning-based inter-and intra-application thermal optimization for lifetime improvement of multicore systems. In: Proceedings of the 51st Annual Design Automation Conference, pp 1–6

Das A, Kumar A, Veeravalli B (2015a) Reliability and energy-aware mapping and scheduling of multimedia applications on multiprocessor systems. IEEE Trans Parallel Distrib Syst (TPDS) 27(3):869–884

Das A, Kumar A, Veeravalli B, Shafik R, Merrett G, Al-Hashimi B (2015b) Workload uncertainty characterization and adaptive frequency scaling for energy minimization of embedded systems. In: Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, pp 43–48

Das AK, Kumar A, Veeravalli B, Catthoor F (2018) Introduction. Springer International Publishing, Cham, pp 1–21. https://doi.org/10.1007/978-3-319-69374-3_1

Dennard RH, Gaensslen FH, Rideout VL, Bassous E, LeBlanc AR (1974) Design of ion-implanted MOSFET's with very small physical dimensions. IEEE J Solid-State Circuits 9(5):256–268. https://doi.org/10.1109/JSSC.1974.1050511

Dey S, Singh AK, Wang X, McDonald-Maier K (2020) User interaction aware reinforcement learning for power and thermal efficiency of CPU-GPU mobile MPSoCs. In: 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, pp 1728–1733

Dinakarrao SMP, Joseph A, Haridass A, Shafique M, Henkel J, Homayoun H (2019) Application and thermal-reliability-aware reinforcement learning based multi-core power management. ACM J Emerg Technol Comput Syst (JETC) 15(4):1–19

Flautner K, Kim NS, Martin S, Blaauw D, Mudge T (2002) Drowsy caches: simple techniques for reducing leakage power. In: Proceedings of the 29th Annual International Symposium on Computer Architecture, ISCA '02. IEEE Computer Society, USA, pp 148–157

Gnad D, Shafique M, Kriebel F, Rehman S, Sun D, Henkel J (2015) Hayat: Harnessing dark silicon and variability for aging deceleration and balancing. In: 2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC). IEEE, pp 1–6

Held J, Bautista J, Koehl S (2006) From a few cores to many: a tera-scale computing research overview. White paper, Intel

Henkel J, Dutt N (2021) Dependable embedded systems. Springer Nature. https://link.springer.com/book/10.1007/978-3-030-52017-5

Huang K, Ha Y, Zhao R, Kumar A, Lian Y (2014) A low active leakage and high reliability phase change memory (PCM) based non-volatile FPGA storage element. IEEE Trans Circuits Syst I: Regul Pap 61(9):2605–2613

Isuwa S, Dey S, Ortega AP, Singh AK, Al-Hashimi BM, Merrett GV (2022) QUAREM: Maximising QoE through adaptive resource management in mobile MPSoC platforms. ACM Trans Embed Comput Syst (TECS) 21(4):1–29

Jin S, Qie X, Hao S (2019) Virtual machine allocation strategy in energy-efficient cloud data centres. Int J Commun Netw Distrib Syst 22(2):181–195

Khanh PN, Singh AK, Kumar A, Aung KMM (2013) Incorporating energy and throughput awareness in design space exploration and run-time mapping for heterogeneous MPSoCs. In: 2013 Euromicro Conference on Digital System Design. IEEE, pp 513–521

Kim N, Austin T, Baauw D, Mudge T, Flautner K, Hu J, Irwin M, Kandemir M, Narayanan V (2003) Leakage current: Moore's law meets static power. Computer 36(12):68–75. https://doi.org/10.1109/MC.2003.1250885

Kim T, Sun Z, Chen HB, Wang H, Tan SXD (2017) Energy and lifetime optimizations for dark silicon manycore microprocessor considering both hard and soft errors. IEEE Trans Very Large Scale Integr (VLSI) Syst 25(9):2561–2574. https://doi.org/10.1109/TVLSI.2017.2707401

Kim W, Gupta MS, Wei GY, Brooks D (2008) System level analysis of fast, per-core DVFS using on-chip switching regulators. In: 2008 IEEE 14th International Symposium on High Performance Computer Architecture, pp 123–134. https://doi.org/10.1109/HPCA.2008.4658633

Kontorinis V, Shayan A, Tullsen DM, Kumar R (2009) Reducing peak power with a table-driven adaptive processor core. In: 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp 189–200. https://doi.org/10.1145/1669112.1669137

Kumar R, Zyuban V, Tullsen D (2005) Interconnections in multi-core architectures: understanding mechanisms, overheads and scaling. In: 32nd International Symposium on Computer Architecture (ISCA'05), pp 408–419. https://doi.org/10.1109/ISCA.2005.34

Li B, Wang X, Singh AK, Mak T (2019) On runtime communication and thermal-aware application mapping and defragmentation in 3D NoC systems. IEEE Trans Parallel Distrib Syst 30(12):2775–2789

Li S, Ahn JH, Strong RD, Brockman JB, Tullsen DM, Jouppi NP (2013) The McPAT framework for multicore and manycore architectures: simultaneously modeling power, area, and timing. ACM Trans Archit Code Optim 10(1). https://doi.org/10.1145/2445572.2445577

Lin X, Wang Y, Pedram M (2016) A reinforcement learning-based power management framework for green computing data centers. In: 2016 IEEE International Conference on Cloud Engineering (IC2E). IEEE, pp 135–138

Liu N, Li Z, Xu J, Xu Z, Lin S, Qiu Q, Tang J, Wang Y (2017) A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning. In: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS). IEEE, pp 372–382

Ma Y, Zhou J, Chantem T, Dick RP, Wang S, Hu XS (2018) Online resource management for improving reliability of real-time systems on "big–little" type MPSoCs. IEEE Trans Comput-Aided Des Integr Circuits Syst 39(1):88–100

Ma Y, Zhou J, Chantem T, Dick RP, Wang S, Hu XS (2019) Improving reliability of soft real-time embedded systems on integrated CPU and GPU platforms. IEEE Trans Comput-Aided Des Integr Circuits Syst 39(10):2218–2229

Mahesri A, Vardhan V (2005) Power consumption breakdown on a modern laptop. In: Falsafi B, VijayKumar TN (eds) Power-aware computer systems. Springer, Berlin/Heidelberg, pp 165–180

Mamun SA (2021) Exploring wireless data center networks: can they reduce energy consumption while providing secure connections? Ph.D. thesis, Rochester Institute of Technology, Rochester

Mamun SA, Gilday A, Singh AK, Ganguly A, Merrett GV, Wang X, Al-Hashimi BM (2020) Intra- and inter-server smart task scheduling for profit and energy optimization of HPC data centers. J Low Power Electron Appl 10(4):32

Medina R, Borde E, Pautet L (2018) Availability enhancement and analysis for mixed-criticality systems on multi-core. In: Proceedings on Design, Automation & Test in Europe Conference & Exhibition (DATE), pp 1271–1276

Mittal S (2012) A survey of architectural techniques for dram power management. Int J High Perform Syst Archit 4(2):110–119. https://doi.org/10.1504/IJHPSA.2012.050990

Munawar W, Khdr H, Pagani S, Shafique M, Chen JJ, Henkel J (2014) Peak power management for scheduling real-time tasks on heterogeneous many-core systems. In: Proceedings of IEEE International Conference on Parallel and Distributed Systems (ICPADS), pp 200–209. https://doi.org/10.1109/PADSW.2014.7097809

Narayanan S, Sartori J, Kumar R, Jones DL (2010) Scalable stochastic processors. In: 2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010), pp 335–338. https://doi.org/10.1109/DATE.2010.5457181

Navardi M, Ranjbar B, Rohbani N, Ejlali A, Kumar A (2022) Peak-Power Aware Life-Time Reliability Improvement in Fault-Tolerant Mixed-Criticality Systems. IEEE Open J Circuits Syst 3:199–215. https://doi.org/10.1109/OJCAS.2022.3207598

Nawathe UG, Hassan M, Yen KC, Kumar A, Ramachandran A, Greenhill D (2008) Implementation of an 8-core, 64-thread, power-efficient sparc server on a chip. IEEE J Solid-State Circuits 43(1):6–20. https://doi.org/10.1109/JSSC.2007.910967

Nicolaescu D, Veidenbaum A, Nicolau A (2003) Reducing data cache energy consumption via cached load/store queue. In: Proceedings of the 2003 International Symposium on Low Power Electronics and Design, ISLPED'03. pp 252–257. https://doi.org/10.1109/LPE.2003.1231871

Oudaa T, Gharsellaoui H, Ahmed SB (2021) An agent-based model for resource provisioning and task scheduling in cloud computing using DRL. Proc Comput Sci 192:3795–3804

Pagani S, Pathania A, Shafique M, Chen JJ, Henkel J (2016) Energy efficiency for clustered heterogeneous multicores. IEEE Trans Parallel Distrib Syst (TPDS) 28(5):1315–1330

Pagani S, Chen JJ, Shafique M, Henkel J (2018) Advanced techniques for power, energy, and thermal management for clustered manycores. Springer. https://link.springer.com/book/10.1007/978-3-319-77479-4

Pathania A, Pagani S, Shafique M, Henkel J (2015) Power management for mobile games on asymmetric multi-cores. In: Proceedings of IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED). IEEE, pp 243–248

PD SM, Lin J, Zhu S, Yin Y, Liu X, Huang X, Song C, Zhang W, Yan M, Yu Z, et al. (2017) A scalable network-on-chip microprocessor with 2.5D integrated memory and accelerator. IEEE Trans Circuits Syst I: Regul Pap 64(6):1432–1443

Puttaswamy K, Loh GH (2007) Thermal herding: Microarchitecture techniques for controlling hotspots in high-performance 3D-integrated processors. In: 2007 IEEE 13th International Symposium on High Performance Computer Architecture, pp 193–204. https://doi.org/10.1109/HPCA.2007.346197

Ranjbar B, Nguyen TDA, Ejlali A, Kumar A (2019) Online peak power and maximum temperature management in multi-core mixed-criticality embedded systems. In: Proceedings of Euromicro Conference on Digital System Design (DSD), pp 546–553. https://doi.org/10.1109/DSD.2019.00084

Ranjbar B, Nguyen TDA, Ejlali A, Kumar A (2021) Power-aware run-time scheduler for mixed-criticality systems on multi-core platform. IEEE Trans Comput-Aided Des Integr Circuits Syst (TCAD) 40(10):2009–2023. https://doi.org/10.1109/TCAD.2020.3033374

Ranjbar B, Hosseinghorban A, Salehi M, Ejlali A, Kumar A (2022) Toward the design of fault-tolerance-and peak-power-aware multi-core mixed-criticality systems. IEEE Trans Comput-Aided Des Integr Circuits Syst (TCAD) 41(5):1509–1522. https://doi.org/10.1109/TCAD.2021.3082495

Rodrigues R, Annamalai A, Koren I, Kundu S, Khan O (2011) Performance per watt benefits of dynamic core morphing in asymmetric multicores. In: 2011 International Conference on Parallel Architectures and Compilation Techniques, pp 121–130. https://doi.org/10.1109/PACT.2011.18

Sahoo SS, Kumar A (2021a) CLEO-CoDE: Exploiting constrained decoding for cross-layer energy optimization in heterogeneous embedded systems. In: 2021 IFIP/IEEE 29th International Conference on Very Large Scale Integration (VLSI-SoC), pp 1–6. https://doi.org/10.1109/VLSI-SoC53125.2021.9606983

Sahoo SS, Kumar A (2021b) Using Monte Carlo tree search for EDA – a case-study with designing cross-layer reliability for heterogeneous embedded systems. In: 2021 IFIP/IEEE 29th International Conference on Very Large Scale Integration (VLSI-SoC), pp 1–6. https://doi.org/10.1109/VLSI-SoC53125.2021.9606987

Sahoo SS, Veeravalli B, Kumar A (2016) Cross-layer fault-tolerant design of real-time systems. In: 2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), pp 63–68. https://doi.org/10.1109/DFT.2016.7684071

Sahoo SS, Veeravalli B, Kumar A (2018) CLRFrame: an analysis framework for designing cross-layer reliability in embedded systems. In: 31st International Conference on VLSI Design and 17th International Conference on Embedded Systems, VLSID 2018, 6–10 Jan 2018, Pune, India, pp 307–312. https://doi.org/10.1109/VLSID.2018.81, http://doi.ieeecomputersociety.org/10.1109/VLSID.2018.81

Sahoo SS, Veeravalli B, Kumar A (2019) A hybrid agent-based design methodology for dynamic cross-layer reliability in heterogeneous embedded systems. In: Design Automation Conference, DAC 2019, 2–6 June 2019, Las Vegas, Nevada

Sahoo SS, Veeravalli B, Kumar A (2020) CL(R)early: An early-stage DSE methodology for cross-layer reliability-aware heterogeneous embedded systems. In: 2020 57th ACM/IEEE Design Automation Conference (DAC), pp 1–6. https://doi.org/10.1109/DAC18072.2020.9218747

Sahoo SS, Kumar A, Decky M, Wong SCB, Merrett GV, Zhao Y, Wang J, Wang X, Singh AK (2021a) Emergent design challenges for embedded systems and paths forward: mixed-criticality, energy, reliability and security perspectives. In: Proceedings of the 2021 International Conference on Hardware/Software Codesign and System Synthesis, CODES/ISSS '21. Association for Computing Machinery, New York, pp 1–10. https://doi.org/10.1145/3478684.3479246

Sahoo SS, Ranjbar B, Kumar A (2021b) Reliability-aware resource management in multi-/many-core systems: a perspective paper. J Low Power Electron Appl 11(1):7

Salehi M, Ejlali A (2014) A hardware platform for evaluating low-energy multiprocessor embedded systems based on cots devices. IEEE Trans Ind Electron (TIE) 62(2):1262–1269

Shafique M, Khan MUK, Tüfek O, Henkel J (2015) EnAAM: energy-efficient anti-aging for on-chip video memories. In: Proceedings of the 52nd Annual Design Automation Conference, pp 1–6

Sheikh SZ, Pasha MA (2018) Energy-efficient multicore scheduling for hard real-time systems: a survey. ACM Trans Embed Comput Syst 17(6). https://doi.org/10.1145/3291387

Shekarisaz M, Hoseinghorban A, Bazzaz M, Salehi M, Ejlali A (2021) MASTER: Reclamation of hybrid scratchpad memory to maximize energy saving in multi-core edge systems. IEEE Trans Sustain Comput

Singh AK, Das A, Kumar A (2013) Energy optimization by exploiting execution slacks in streaming applications on multiprocessor systems. In: Proceedings of the Design Automation Conference (DAC), pp 1–7

Singh AK, Dziurzanski P, Indrusiak LS (2015a) Market-inspired Dynamic Resource Allocation in Many-core High Performance Computing Systems. In: IEEE International Conference on High Performance Computing & Simulation (HPCS), pp 413–420

Singh AK, Dziurzanski P, Indrusiak LS (2015b) Value and energy optimizing dynamic resource allocation in many-core HPC systems. In: 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom). IEEE, pp 180–185

Singh AK, Dziurzanski P, Indrusiak LS (2016a) Value and energy aware adaptive resource allocation of soft real-time jobs on many-core HPC data centers. In: 2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC). IEEE, pp 190–197

Singh AK, Shafique M, Kumar A, Henkel J (2016b) Analysis and mapping for thermal and energy efficiency of 3-D video processing on 3-D multicore processors. IEEE Trans Very Large Scale Integr (VLSI) Syst 24(8):2745–2758

Singh AK, Dey S, McDonald-Maier K, Basireddy KR, Merrett GV, Al-Hashimi BM (2020) Dynamic energy and thermal management of multi-core mobile platforms: a survey. IEEE Design & Test 37(5):25–33

Sun G, Liao D, Zhao D, Xu Z, Yu H (2015) Live migration for multiple correlated virtual machines in cloud-based data centers. IEEE Trans Services Comput 11(2):279–291

Turakhia Y, Raghunathan B, Garg S, Marculescu D (2013) Hades: architectural synthesis for heterogeneous dark silicon chip multi-processors. In: 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC), pp 1–7. https://doi.org/10.1145/2463209.2488948

Walker MJ, Merrett GV, Al-Hashimi B (2019) Power modelling of multicore systems. https://doi.org/10.1049/PBPC022E_ch13, https://digital-library.theiet.org/content/books/10.1049/pbpc022e_ch13

Weste NH, Harris D (2015) CMOS VLSI design: a circuits and systems perspective. Pearson Education India

Zhu Z, Zhang W, Chaturvedi V, Singh AK (2020) Energy minimization for multicore platforms through DVFS and VR phase scaling with comprehensive convex model. IEEE Trans on Comput-Aided Des Integr Circuits Syst 39(3):686–699. https://doi.org/10.1109/TCAD.2019.2894835

General-Purpose Multicore Architectures

18

Saugata Ghose

Contents

Introduction . . . 596
Motivating the Need for Concurrent Processing . . . 598
  Classifying Parallel Computing Hardware . . . 598
  Multiprocessing . . . 599
  Thread-Level Parallelism Within an Application . . . 601
  What to Do With All These Transistors? . . . 604
Multicore CPU Hardware Design . . . 606
  Optimizing CPU Cores for Parallelism . . . 606
  Sharing Caches and Main Memory . . . 609
  Coordinating Memory Requests Across Cores . . . 614
  Scaling to Many Cores . . . 614
Managing Memory . . . 615
  Shared-Memory Model . . . 617
  Main Memory Policies . . . 618
  Mitigating Interference . . . 621
  Cache Coherence . . . 622
  Memory Consistency Models . . . 626
Optimizing Operating Systems for Multicore CPUs . . . 628
Evaluating Multicore CPUs . . . 631
The Evolution of Multicore CPUs . . . 635
  Systems-on-Chip . . . 635
  Heterogeneous CPU Cores . . . 637
  Chiplet-Based Multicore Design . . . 638
Conclusion . . . 640
References . . . 640

S. Ghose (✉)
University of Illinois Urbana-Champaign, Urbana, IL, USA
e-mail: ghose@illinois.edu

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_46

Abstract

The first years of the 2000s led to an inflection point in computer architectures: While the number of available transistors on a chip continued to grow, crucial transistor scaling properties started to break down and result in increasing power consumption, while aggressive single-core performance optimizations were resulting in diminishing returns due to inherent limits in instruction-level parallelism. This led to the rise of multicore CPU architectures, which are now commonplace in modern computers at all scales. This chapter discusses the evolution of multicore CPUs since their introduction. Starting with a historic overview of multiprocessing, the chapter explores the basic microarchitecture of a multicore CPU, key challenges resulting from shared memory resources, operating system modifications to optimize multicore CPU support, popular metrics for multicore evaluation, and recent trends in multicore CPU design.

Keywords

Multicore CPU · Chip multiprocessor · Parallel computer architecture · System-on-chip (SoC) · Thread-level parallelism (TLP)

Introduction

From the first commercial microprocessors in the 1970s through the end of the 1990s, microprocessors available on the market typically consisted of a single CPU core per chip. During that time, significant architectural advancements were made to improve the performance of the CPU core, including (but not limited to) techniques such as out-of-order processing and superscalar execution that extracted *instruction-level parallelism* (ILP) from single-thread sequential programs. These architectural advancements were driven by two trends that governed advances in semiconductor manufacturing process technologies. The first, Moore's Law (Moore 1965), was an observation made in 1965 by Gordon Moore that the number of transistors on an integrated circuit (IC) doubled every year, which he revised in 1975 (Moore 1975) to forecast a doubling every 2 years after 1980. Figure 1 shows a progression of Moore's Law, using real CPUs as examples, between 1971 and 2024. The second, Dennard Scaling (Dennard et al. 1972, 1974), was a relationship identified by Robert Dennard and his colleagues at IBM in the early 1970s, identifying that with every new manufacturing process technology node (an approximately 18-month interval at the time), both the area and the power consumption of a single transistor were half of what was observed in the previous generation.

As Moore's Law and Dennard Scaling made it more economical to increase the number of transistors on a chip (effectively providing double the number of transistors, at the same area and power budget, every 18–24 months), manufacturers dedicated these additional transistors toward increasing the performance of the single CPU core. Unfortunately, two critical factors made it difficult to keep

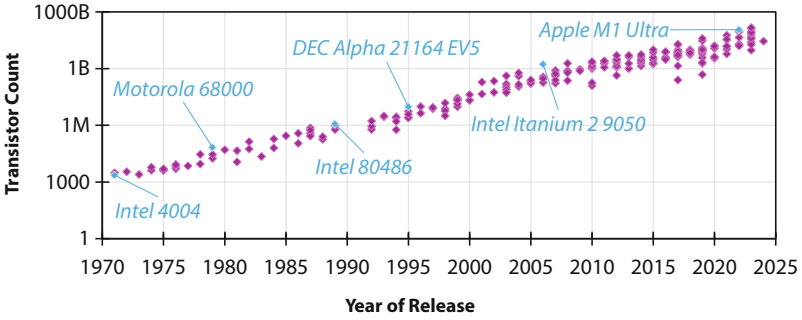


Fig. 1 Log–linear plot of selected CPUs introduced between 1971 and 2024, illustrating the progression of Moore's Law

continuing this trend. First, as the "free" rewards of scaling started to break down in the early 2000s, the areal power density (directly correlated to the amount of heat dissipated per unit area) of high-end CPUs began growing rapidly (De and Borkar 1999; Frank et al. 2001). Then-contemporary projections estimated that if single-core CPU development continued along the trends of the time, commonplace passive cooling elements would no longer be able to dissipate the heat generated by the CPU (De and Borkar 1999). Second, more aggressive techniques for ILP were yielding diminishing returns, requiring high hardware costs for meager performance benefits (Ronen et al. 2001). These factors forced manufacturers to reconsider how to use additional transistors to continue to deliver performance improvements, now in a power-efficient manner (De and Borkar 1999; Ronen et al. 2001; Esmaeilzadeh 2011).

This reconsideration led to the widespread adoption of *multicore CPUs* (also known as *chip multiprocessors*) (Ronen et al. 2001; Esmaeilzadeh 2011): Instead of trying to make a single CPU core more powerful, manufacturers now implement multiple, simpler CPU cores within a single chip that can run multiple tasks concurrently. (This chapter uses the term *concurrent processing* to refer strictly to multiple independent tasks executing at the same time. It refers to the time multiplexing of a CPU core across multiple tasks as *time-sharing*.) The simpler core designs significantly reduced the areal power density (e.g., W/mm^2) of the CPUs. Multicore CPUs perform concurrent processing by (1) performing concurrent *multiprocessing* of more than one program and/or (2) extracting *thread-level parallelism* (TLP) from a parallelizable application. While initial commercial multicore CPUs started out with two identical CPU cores, today's multicore CPUs have a wide range of configurations, with some containing dozens of cores.

This chapter will examine six topics related to multicore CPU architectures. First, it will motivate the benefits and limitations of processing multiple tasks concurrently and how they drove the need for parallel processing. Second, it will examine the hardware design of a typical multicore CPU and the changes required to support the efficient execution of multiple programs on multiple CPU cores. Third, it will

study how memory management policies change to handle the increased traffic from multiple cores. Fourth, it will address software issues that optimize the use of multicore CPUs. Fifth, it will introduce common metrics that are used in the context of multicore CPUs. Finally, it will close by briefly discussing how commercial multicore CPUs have evolved since their introduction.

Motivating the Need for Concurrent Processing

As early as the days of analog computing, the benefits of parallelizing tasks became a clear goal: Luigi Federico Menabrea, in his 1842 study of Charles Babbage's Analytical Engine (Menabrea 1842), noted that a key advantage of mechanized computation would be its ability to produce several results at the same time. In the mid-twentieth century, as digital computers evolved rapidly, there was a pressing need to maximize both the performance and utilization of these computers, and the extraction of parallelism, in its various forms, became a key approach to meet these needs.

The early decades of electronic computing saw the introduction of several now-commonplace parallelization techniques, such as instruction pipelining (1941, with the Zuse Z3 (Rojas 1996; Zuse 1949), followed by significant advances in 1961 with the IBM 7030 Stretch (Dunwell 1956) and ILLIAC II (Taub et al. 1957) supercomputers), superscalar processing (1964, with the CDC 6600 supercomputer (Thornton 1964)), and general-purpose time-sharing (1961, with the Compatible Time-Sharing System operating system (Corbató et al. 1962)). (In this section, significant efforts were made to identify milestone systems that introduced or made critical leaps forward in now-fundamental parallelism techniques. However, given the rapid pace of concurrent computer development in the 1950s and 1960s, combined with limited available historical resources, these examples may not always be the true originators of the techniques. When possible, dates associated with systems denote the year of first delivery, as a proxy for the completion of the first working system.) This set the stage for the rise of the two key types of parallelism exposed by multicore CPUs: multiprocessing and thread-level parallelism. While both of these types of parallelism can be exploited by a range of hardware organizations, historical trends in chip manufacturing process technologies drove the emergence of the multicore CPU.

Classifying Parallel Computing Hardware

Given the rise of multiple parallel computing techniques throughout the 1960s, there was a need to categorize the techniques based on broadly shared principles. To this end, Michael Flynn developed what is now known as Flynn's taxonomy in 1966 (Flynn 1966). The original taxonomy classified computer architectures into four categories, along two dimensions as shown in Fig. 2:

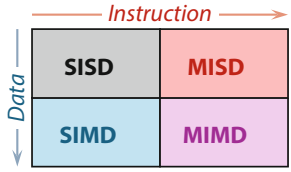


Fig. 2 An illustration of Flynn's taxonomy

- Single instruction stream, single data stream (SISD)
- Single instruction stream, multiple data stream (SIMD)
- Multiple instruction stream, single data stream (MISD)
- Multiple instruction stream, multiple data stream (MIMD)

Under Flynn's taxonomy, non-pipelined, non-superscalar single-core CPUs that employ a von Neumann architecture are a canonical example of an SISD architecture. Examples of SIMD architectures include graphics processing units (GPUs), array processors, and vector extensions for CPUs. One reasonably accepted example of an MISD architecture is redundant computing, such as the systems employed in avionics units. (There is some latent controversy about which category pipelined CPUs fall into. In its best imitation of Switzerland and its neutrality, this chapter does not take a stand for any category. Sorry for being so bland.) The MIMD category includes most computers that are capable of multiprocessing (discussed in the "Multiprocessing" section), including computers with multicore CPUs.

In a multicore CPU, each CPU core is capable of executing one or more independent streams of instructions, with each instruction stream operating on its own data set (though this data set can potentially overlap with the data sets being used by other cores). To support this MIMD model of execution, each core has its own control logic, and much of the computation takes place independent from other cores. The section titled "Multicore CPU Hardware Design" elaborates on this more while also discussing when coordination between cores does occur (both explicitly and implicitly).

Multiprocessing

Before the days of personal computers (PCs), mainframes were the dominant form factor of computers. Mainframe computers had a relatively high cost: as an example, the IBM System/360 Model 25, a low-end variant of IBM's highly popular mainframes, was announced in January 1968 with a purchase price of US $253,000 (US $2.33 million in 2024 dollars), with a monthly rental option at $5,330 (US $49,000 in 2024 dollars) (Whitney and White 1968). Given the high demand to use these computers, batching and time-sharing (i.e., time multiplexing) systems became commonplace. Systems capable of batching would queue up multiple jobs to execute on a computer back-to-back, while time-sharing systems extended this capability to execute multiple programs on the same computer through temporal multiplexing.

More concretely, time-sharing involves the use of *scheduling quanta*, which are (typically predetermined) periods of time that a program can execute for before being *preempted* (i.e., switched out). A time-sharing system runs one program for a single scheduling quantum, after which it switches out the program from the CPU and looks at a queue of pending programs (which include the one just switched out, assuming it did not complete within the quantum). The system then selects the next program to run, switches it into the CPU, and lets it execute for a single scheduling quantum, before repeating the switch-out/switch-in procedure. Time-sharing allowed multiple users to interact with a single computer at the same time, allowing each user's program to make forward progress even though the computer had only a single CPU. Thanks to the typically short scheduling quanta (on the order of milliseconds in modern machines), time-sharing often gives each user the illusion that the computer is continuously executing their application, as the scheduling quantum is significantly faster than human perception of response time.

While time-sharing represents a key technological shift in the accessibility of computers, the technique suffers from two critical overheads that significantly extend the overall execution time of a program. First, in an ideal machine, each program now takes longer to complete as it repeatedly waits on other programs to execute. For example, if a computer runs ten programs, each taking the same amount of time to finish, and the operating system uses round-robin scheduling to switch between each of the programs, the programs will take 10× longer to complete than if they ran uninterrupted. Second, in real-world machines, the execution time of each program is further increased by the overhead of *context switching*. When a program is switched out, its registers (and for some machines, dirty cache values) must be saved somewhere, and these values must be restored when the program is switched back in.

As a natural progression toward concurrent execution without these overheads, researchers began to explore whether a computer could incorporate multiple CPUs, where the CPUs have the ability to communicate with each other. (Alternative techniques such as multithreading were also being developed around this time, with 1960's Bull Gamma 60 (Dreyfus 1958) and Honeywell 800 (Minneapolis–Honeywell DATAmatic Division 1960) being two early examples of hardware multithreading support. A constrained version of multithreading was implemented in the DYSEAC in 1954 (Leiner 1952).) This technique, known as *multiprocessing*, was first implemented in the Burroughs B 5000 (Longergan and King 1961) and D825 (Anderson et al. 1962) mainframe computers, released in 1961 and 1962, respectively. In a computer capable of multiprocessing, multiple programs can execute simultaneously, without the need to time-share the CPU. An operating system (OS) typically sees a multi-CPU computer as a single system, and a multiprocessing-capable OS is responsible for assigning programs to specific CPUs. Given the high cost of CPUs, many multi-CPU machines still perform time-sharing on each CPU in addition to multiprocessing, and the OS uses various scheduling heuristics to determine how best to choose which CPU a program should be scheduled on.

Thread-Level Parallelism Within an Application

While multiprocessing enabled increases in the throughput and/or productivity of a machine by allowing many applications to execute concurrently there still remained a need to reduce the execution time of a single application. Hardware techniques such as instruction pipelining and superscalar execution, along with compiler-assisted techniques such as very long instruction words (VLIW), unlocked ILP within an application, by identifying independent instructions that can execute concurrently. While ILP techniques have provided significant performance boosts within a single process, there was also a need to exploit multiprocessing-capable machines to run a *parallel program*, where multiple constituent tasks from the program execute concurrently.

In order for a parallel program to be decomposed into multiple parallelizable tasks, there needs to be (1) some way for these tasks to execute concurrently for much of their lifetime and (2) some way for the tasks to communicate and synchronize with each other in order to generate a single set of results. Parallel programs require support at both a software and hardware level to execute. In software, the core concepts on how to decompose a program into multiple parallel tasks were developed in the 1950s and 1960s, and Dekker's algorithm from 1960 (Dijkstra 1962, 1968; Dekker 2022) allowed programmers to enable mutually exclusive access to shared memory locations among two concurrent processes. In hardware, the development of early multi-CPU machines was accompanied with the introduction of shared memory. For example, the Burroughs B 5000 from 1962 (Longergan and King 1961) provided primitive shared memory support by letting all of its CPUs have shared access to its memory modules. As the concept of parallel programming evolved, these tasks eventually became formalized as *threads* that can execute asynchronously (Witt 1966), and the use of multiple concurrent threads to accelerate the execution of a program became known as TLP.

The performance improvement that a parallel program can achieve over its sequential counterpart depends primarily on two factors: (1) the number of threads that can be executed concurrently and (2) the amount of synchronization needed. Amdahl's Law models the impact of the thread count on maximum performance improvements for a *fixed* problem size (Amdahl 1967). The law states that the theoretical *speedup* S (i.e., the reduction in execution time) of a parallel program over its sequential counterpart can be expressed as

$$S = \frac{1}{(1-f) + \frac{f}{n}} \tag{1}$$

where f is the fraction of the program that can be parallelized, and n is the number of concurrent threads that can execute the parallelizable part of the program. To visualize Amdahl's Law, Fig. 3 shows an example where $f = 0.4$ (i.e., 40% of the program can be parallelized), for various thread counts. As the figure shows, for

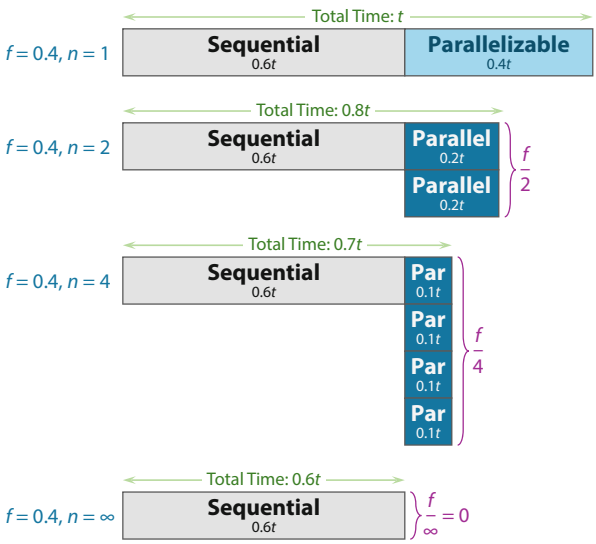


Fig. 3 Amdahl's Law visualized for an example application

every doubling of the thread count, the reduction in total time is only half of the reduction achieved by the previous doubling.

Figure 4a shows the theoretical parallel speedup achievable according to the law, for different values of f. A key takeaway from Amdahl's Law is that even for an infinite number of concurrent threads (i.e., $n = \infty$), the maximum speedup S of a parallel program is bound by the time it takes to execute the part of the program that is *not* parallelizable. While Amdahl's Law encompasses key properties of parallel execution, it has two important limitations.

First, Amdahl's Law does not explicitly account for the additional overhead of synchronization primitives. As more threads execute concurrently, the contention for acquiring mutually exclusive access to a portion of shared memory can increase. For example, if all of the threads of a parallel program share and update a single counter, they must use a mutual exclusion primitive (i.e., *mutex*) to ensure that updates from one thread are not inadvertently lost by another thread. This requires each thread to acquire the mutex whenever it updates the counter, and other threads that attempt to acquire a mutex that is currently held by another thread will stall. As more threads execute concurrently, the likelihood increases that more threads will contend at the same time to acquire the mutex, introducing more stalls. In practice, these synchronization overheads from mutex contention (as well as interference between threads due to resource sharing; see "Sharing Caches and Main Memory") can make it such that adding a parallel thread can actually *decrease* the parallel speedup, as shown in Fig. 5.

Second, Amdahl's Law assumes that the problem size is fixed regardless of the number of parallel threads. In reality, the number of threads used to parallelize a program is typically linked with the number of available CPUs. For a multiprocessing machine, when there are more CPUs, a program has more resources available

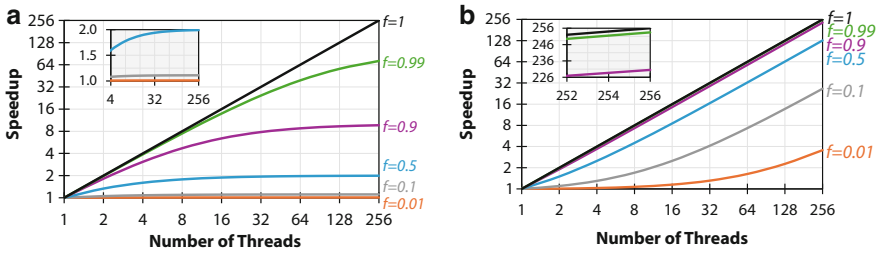


Fig. 4 Comparison of theoretical parallel speedup estimated by Amdahl's Law and Gustafson's Law, for different parallelizable fractions f. Inset graphs show a zoomed-in section of the main graph for clarity. (**a**) Amdahl's Law. (**b**) Gustafson's Law

to it, such as memory capacity. As a result, programmers and/or users can increase the problem size to take advantage of these extra resources. Since Amdahl's Law does not capture the impact of these extra resources, it can provide a pessimistic estimate of the capabilities of a multi-CPU machine. Gustafson's Law (Gustafson 1988) accounts for this, by calculating the parallel speedup S as

$$S = (1 - f) + f \times n \tag{2}$$

Figure 4b shows the predicted speedups from the law, to provide a comparison to Amdahl's Law.

Real-world parallel system performance deviates from both Amdahl's Law and Gustafson's Law. For example, while one could treat Gustafson's Law as an optimistic upper bound on performance, real machines can sometimes outperform the estimated parallel speedup from Gustafson's Law due to data sharing. If one thread brings data into a shared memory that is subsequently used by a second thread (exploiting temporal and/or spatial locality across threads), the second thread no longer pays the memory latency required for that subsequent data access (assuming that the data is not evicted). This phenomenon is an example of *superlinear speedup*.

Putting this all together, Fig. 5 shows a synthetically constructed example of the parallel speedup one could expect to observe on a real parallel system. The example assumes that 99% of the application can be parallelized across all threads, while the remaining 1% must execute sequentially. It also assumes that the machine has 32 CPU cores, and since it has a fixed set of available resources, Amdahl's Law can be used to predict performance. There are three observations from the figure. First, even with a single thread, the observed speedup is less than 1. This is because parallel speedup is compared against the best sequential implementation (see "Evaluating Multicore CPUs"), and a parallel implementation (with a configurable parameter n that sets the thread count) typically has overheads associated with adding code to support parallel execution, and these overheads are observed when $n = 1$. Second, superlinear speedups can even exceed perfect parallelism. As mentioned above, this is the result of threads helping each other through data locality. Third, after

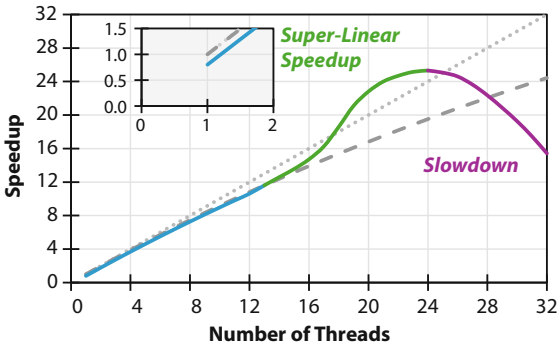


Fig. 5 A synthetic example of observable behavior in a real-world parallel system. The dotted line shows ideal parallelism ($f = 1$ with Amdahl's Law), the dashed line shows expected performance ($f = 0.99$ with Amdahl's Law), and the solid line shows observed performance

a certain thread count (24 in our example), the performance at a higher thread count is actually *lower* than the performance at a lower thread count. This is because the benefits of additional thread-level parallelism and additional data locality are overcome by synchronization and interference overheads.

Note that while Fig. 5 is one example of parallel application behavior, its trends are not universal. As one example, some applications exhibit what is known as *embarrassingly parallel* behavior (Mattson et al. 2004), where an application continues to achieve near-ideal parallel speedup even at high thread counts, due to limited need for synchronization and serialization.

What to Do With All These Transistors?

Through the 1990s (and drawing some very broad, and likely inaccurate, generalizations), the vast majority of parallel computing architectures catered to the large-scale computing market, with much of the effort focused on supercomputers. Most personal computers (PCs) and workstations incorporated only a single-core CPU (or in the case of some high-end workstations, multiple single-core CPUs installed on a multi-socket motherboard). (At that time, there was a distinction between more mainstream PCs and high-performance workstations that targeted power users. Today, in large part due to advancements in PC capabilities throughout the 1990s, this distinction has mostly disappeared, but the chapter references it here for historical context.) Much of the focus on architectural innovation remained on improving the performance of the single core, leading to the maturation and widespread commercial adoption of techniques such as out-of-order processing and superscalar execution.

Concurrently, several architects started envisioning a new direction of CPU design. In the early 1990s, with no immediate end in sight to Moore's Law and Dennard Scaling, a number of works speculated that with millions of additional transistors becoming available on a chip within the next decade, it would now be feasible to *replicate multiple CPU cores within a single chip*. This concept came to be known as a *single-chip multiprocessor* (which ultimately led to the

term *chip multiprocessor* or CMP). A number of varying designs were proposed, ranging from asymmetric and/or specialized cores (Joyce et al. 1987; Schmidt and Caesar 1991) to clustered CPUs (Sohi et al. 1995; Fillo et al. 1995), as well as an early symmetric core design with a shared cache (Hanawa et al. 1991). The Hydra CMP project at Stanford started to explore the trade-offs of using *shared caches* across multiple smaller CPU cores for coordination, in comparison to a high-performance single-core CPU, which led to their seminal 1996 paper (Olukotun et al. 1996) that advocated for many of the key features of a modern multicore CPU. Around the same time, architects at IBM started designing what would become the first commercially available multicore CPU for non-embedded computers, the POWER4 (Tendler et al. 2002). (The earliest known implementation of a commercial multicore single-chip CPU is the COP2440 series by National Semiconductor (National Semiconductor Corp. 1982) (along with the COP2404 variant for prototyping), which was released in 1982. The COP2440 combined two embedded CPU cores on a single die to enable concurrent processing of real-time operations, with a shared memory subsystem.) The POWER4 combined two CPU cores onto a single die, (unfortunately, the term chip has overloaded definitions, so this chapter will use the following: a *die* is a piece of semiconductor material (typically silicon), and a *chip* is an integrated circuit that consists of one or more dies; a chip is typically placed inside a *package*, which allows the chip to be placed in a socket to interact with other chips and electrical components) with a core frequency of 1.1 GHz and a *thermal design point* (TDP) of 115 W.

The wider adoption of multicore CPUs was hastened by two trends observed in the late 1990s. First, a handful of computer engineers started raising the alarm on an impending crisis with power consumption. While Moore's Law (focused on increasing transistor count) remained alive and well (in what was truly a simpler time for the architecturally inclined), they predicted that Dennard Scaling would begin to break down in the near future, which when combined with aggressive architectural changes would significantly increase both the power consumption of a chip and the chip's areal power density (De and Borkar 1999; Frank et al. 2001). As one example, in the span of a decade, Intel saw the TDP of its high-end CPUs increases by an order of magnitude, from 3.5 W for the i486DX-25 (Intel Corp. 1989) (released in 1989) to 34.5 W for the Pentium III 600 (Intel Corp. 1999) (released in 1999). (Conventionally, the thermal design point of a CPU represented its maximum power consumption and served as a direct proxy for the maximum amount of heat that the CPU would dissipate. In today's usage, however, TDP may not account for additional power consumed when turbo modes are engaged, and TDP instead represents the longer term maximum power under non-turbo steady state.) Then-contemporary projections expected the thermal density of a chip to double every 4 years, with the density approaching that of a nuclear reactor by the mid-2000s (De and Borkar 1999; Ronen et al. 2001). While the industry still had increasing transistor counts for a fixed area of silicon, it was becoming more difficult to use all of these transistors to their full capacity without surpassing the capabilities of thermal dissipation techniques. Second, more aggressive techniques for extracting ILP, such as deeper pipelining, larger superscalar widths, and deeper

instruction look-ahead for out-of-order processing, were resulting in diminished returns (Ronen et al. 2001). These limits arose due to a combination of (1) inherent limits to the amount of ILP available in modern applications (due to dependencies and data-dependent control flow) and (2) the quickly increasing complexity of logic required to extract further ILP. Combined, these issues made it increasingly difficult to continue aggressive single-core performance scaling, and multicore CPUs presented an attractive alternative to improving system performance.

As the breakdown of Dennard scaling materialized during the first few years of the 2000s, the power consumption issue was exacerbated significantly. This added pressure led the next wave of multicore CPUs to make a key trade-off to avoid the impending limits of thermal density: They operated each CPU core at a *lower* frequency than their single-core contemporaries, in the hopes that the increased parallel processing capability would make up for reduced single-core performance (Esmaeilzadeh 2011). The section titled "Optimizing CPU Cores for Parallelism" discusses more about the CPU core trade-offs, which allowed architects to harness TLP instead of aggressive ILP to improve performance while keeping power issues under control and opened the floodgates for the proliferation of multicore CPUs.

Multicore CPU Hardware Design

While the term *multicore CPU* may lead one to think that there are significant additions at a microarchitectural level to enable parallel execution, the central design of the cores is often (but not always) *simpler* than the most aggressive single-core CPU microarchitectures released by manufacturers. In fact, the majority of the critical hardware changes to enable multicore CPUs lie *outside* of the CPU core. A central tenet of multicore CPU design is that performance can be attained through parallelism and repetition, as opposed to design complexity.

Figure 6 shows a typical example of the major components found in a multicore CPU. Two or more CPU cores sit within the same chip package, with each core having its own (i.e., *private*) L1 instruction, L1 data, and unified L2 caches. All of the cores have *shared* access to a single *last-level cache* (LLC). The LLC connects with one or more on-chip memory controllers, which provide access to off-chip DRAM modules in the system.

Optimizing CPU Cores for Parallelism

Multicore CPU designers prevent the design and verification of the CPU from increasing linearly with the core count by relying on design reuse through *tiling*. In a homogeneous multicore CPU, the design of the cores is identical, in order to reduce manufacturing and verification/testing costs. (The section titled "The Evolution of Multicore CPUs" discusses popular approaches for heterogeneous multicore CPU design.) Figure 6 shows how the majority of the chip for a typical homogeneous

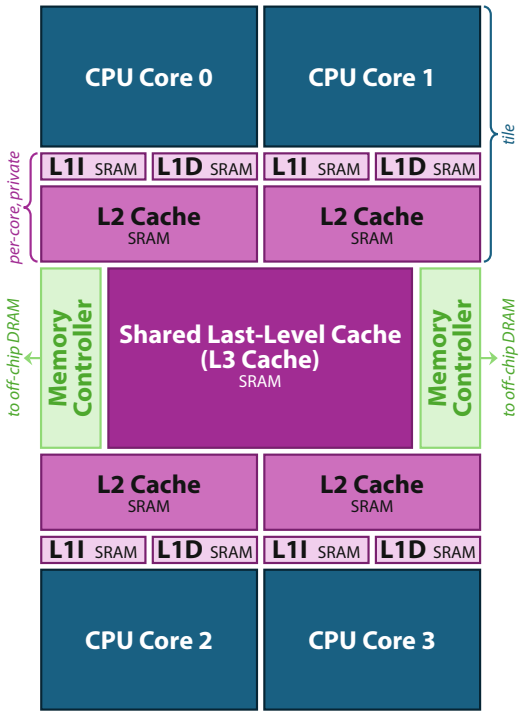


Fig. 6 An example four-core multicore CPU

multicore CPU is made up of multiple tiles, where each tile has exactly the same components. A tile usually consists of a single CPU core and the private caches associated with that core. Some architectures include a slice of the last-level cache in the tile (see the section titled "Sharing Caches and Main Memory"). In addition to these tiles, the chip includes shared components (e.g., controllers that connect the CPU cores to external components, interconnects).

To enable multiple tiles to be stamped out without incurring an exorbitant increase in chip area compared to a powerful single-core CPU, the CPU core within the tile is significantly simplified. As an example, the Core 2 Duo, the second line of desktop multicore CPUs by Intel (and the first whose microarchitecture was designed with dual-core processing in mind), was not based off of the then-state-of-the-art NetBurst microarchitecture (Boggs et al. 2004) that formed the basis of the Pentium 4. Instead, it utilized a new microarchitecture known as Core, which was based on the mobile-oriented Pentium M variant of the P6 microarchitecture (Hinton 2010), with roots spanning back to the mid-1990s.

Examples of CPU Simplification While it is difficult to directly compare CPUs across different microarchitectures, there are three key features that highlight some of the decisions made to simplify the Core 2 Duo's CPU cores compared to the NetBurst microarchitecture, as a motivating example. First, the Core microarchi-

tecture initially consisted of 14 pipeline stages, compared to the 31 pipeline stages found in the last generations of the NetBurst microarchitecture. Second, CPU clock frequencies were significantly lowered as well, with the first generation of Core 2 Duo desktop CPUs achieving a peak frequency of 2.67 GHz (Intel Corp. 2006), compared to a peak frequency of 3.8 GHz for the Pentium 4 desktop CPUs (Intel Corp. 2004). The third is the removal of multithreading support, which allows a CPU to simultaneously execute two applications on the same CPU core through strategic resource sharing. Multithreading was not included in the initial Core 2 Duo CPUs, despite being a prominent feature of the Pentium 4 architecture. It is important, however, to note that these were not long-term changes: deeper pipelines, high clock frequencies, and multithreading support all returned in future Intel multicore CPUs, as continued manufacturing process scaling and microarchitectural innovations enabled these and other features to be implemented more efficiently.

Dynamic Voltage and Frequency Scaling Given the significant implications of the power wall, several low-power techniques were developed around the same time as the emergence of multicore CPUs. One such technique that has become commonplace is *dynamic voltage and frequency scaling* (DVFS) (Macken et al. 1990). With the dramatic performance increase of CPUs through the 1990s, a single CPU was capable of performing significant amounts of computation in a short amount of time. However, this peak throughput is achievable only if there is enough computation to perform. In reality, programs frequently encounter *stalls*, where the CPU is unable to maximize its throughput and can eventually sit idle as it waits for more instructions to be ready to execute. Stalls can occur for a number of reasons, such as having limited opportunities for ILP in a highly pipelined superscalar CPU or having frequent memory or I/O operations that can take hundreds to thousands of cycles to service.

When the CPU is unable to achieve peak throughput, significant energy is wasted by running the CPU at its full voltage and frequency. The dynamic power P_{dyn} consumed by a CPU to execute a program can be modeled as a function of its supply voltage (V) and frequency (f):

$$P_{dyn} \propto \alpha \times C \times V^2 \times f \tag{3}$$

where α is an activity factor, and C is the switched load capacitance of the CPU's transistors. Over time, if the CPU continues at full dynamic power during periods of underutilization, a large amount of energy is wasted. DVFS techniques can detect when the CPU is stalling or being underutilized and reduce the supply voltage and/or the clock frequency of the CPU.

One limitation of DVFS is $V_{\min}$, the minimum operating voltage of the CPU core. A lower bound on $V_{\min}$ is the *threshold voltage* of a transistor, which is the minimum voltage at which the transistor effectively operates as a digital switch. The threshold voltage is a function of the manufacturing process technology. Practically, though, $V_{\min}$ is notably higher than the threshold voltage, in order to ensure the reliable

operation of the CPU. Between the nominal operating voltage (V_{dd}) and $V_{\min}$, DVFS lowers the *voltage* of the selected CPU core. As the voltage is linearly correlated with the CPU core frequency, this results in a cubic reduction in dynamic power, based on Eq. 3. Once DVFS reaches $V_{\min}$, it can continue to lower the frequency but must leave the voltage at $V_{\min}$, resulting in a linear reduction in dynamic power. Figure 7 shows this relationship for a hypothetical CPU, with a base frequency of 4.0 GHz at a TDP of 88 W, V_{dd} = 1.2 V, $V_{\min}$ = 1.0 V, and a turbo boost frequency of 4.4 GHz.

Given that each core in a multicore system has the ability to execute a different thread or workload from the other cores, it is inefficient to manage DVFS settings at a global level. However, early approaches to DVFS relied on large off-chip switching voltage regulators, which each consumed significant area and power and which had response times on the order of tens of microseconds. Within a few years of the introduction of DVFS in commercial systems, on-chip voltage regulators were developed. These on-chip regulators, often called *buck converters* (Kim et al. 2008), are capable of performing sub-100 ns voltage switching, at >90% efficiency. Due to their low cost and fast response, buck converter-based DVFS is now widely used in multicore CPUs, where each CPU core has its own on-chip regulator to allow for per-core DVFS throttling. The voltage and frequency settings for each core are typically selected in the operating system, using a *governor* (see the section titled "Optimizing Operating Systems for Multicore CPUs").

Sharing Caches and Main Memory

At a high level, general-purpose CPU design aims to accommodate a wide range of workloads by providing enough hardware resources to execute most, if not all, of the expected CPU workloads with reasonable performance and efficiency. As a result, CPU design tends to provision resources for close to worst-case behavior while enabling optimizations for the common case. Multicore CPUs provide an opportunity to mitigate some of the costs of worst-case resource allocation in single-

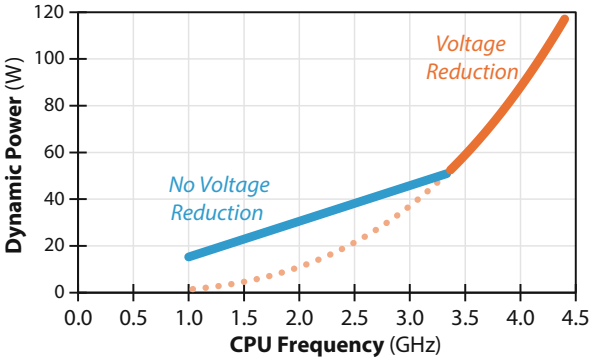


Fig. 7 Dynamic power as a function of DVFS-selected frequency. Solid line shows the actual DVFS behavior, while the dotted line shows idealized further reductions to dynamic power if $V_{\min}$ was not a lower limit to voltage scaling.

core CPUs. For example, in the single-core model, the cache hierarchy should ideally provide enough cache capacity to handle common workloads that exhibit large cacheable data footprints (e.g., workloads with memory access patterns that exploit spatial and/or temporal locality across multiple megabytes of data).

In a multicore CPU, the ability to *share* (i.e., pool together) resources across multiple cores can allow us to support near-worst-case behavior for a subset of cores (e.g., if one core runs a worst-case workload) while keeping *total* resource allocation at a more modest level. Returning to the cache capacity example, while a multicore CPU catering to the worst case (e.g., a situation where each core runs a large-footprint workload) could potentially require several megabytes of cache capacity *per core*, multicore designers instead reduce the per-core amount of cache in the CPU and instead allow cores to share much of their cache capacity with each other. The two most prominent examples of resource sharing in a multicore CPU are in the memory hierarchy: the *last-level cache* (LLC) and main memory.

One issue with sharing resources across CPU cores is *interference* (Bitirgen et al. 2008), where due to the finite amount of available resources, multiple cores may contend with each other for their desired share of these resources. Let us look at a contrived example, with a two-core CPU, where one core is servicing a cache miss that must retrieve data from main memory. As the data for that miss arrives at the LLC, the LLC must evict an existing cache block (assuming that no invalid blocks remain). Interference can occur when the eviction is for a cache block belonging to the other core, as that core will now miss on a subsequent access to the evicted block. Had the first core not triggered the eviction, this subsequent access would have been a cache hit, avoiding the long-latency miss to main memory. The resource contention that results from multicore interference can hurt the effective performance and energy consumption of the CPU (the section titled "Evaluating Multicore CPUs" discusses metrics to quantify this impact). As a result, several approaches have been proposed to mitigate interference in the memory hierarchy (see section "Mitigating Interference").

Multicore Cache Hierarchy A multicore CPU maintains a combination of *private caches* (i.e., caches accessible by only a single CPU core) and *shared caches* (i.e., caches accessible by multiple CPU cores) that are made of *static random-access memory* (SRAM). For each core, a typical modern multicore CPU includes a private L1 instruction cache, a private L1 data cache, and a private L2 unified cache (i.e., a cache that holds both instructions and data). The LLC (the L3 cache in typical modern multicore CPUs) is shared naively (i.e., without any core-based partitioning) across all cores in the CPU: Any workload executing in the CPU can allocate a cache block anywhere in the shared cache. This allows workloads with heterogeneous cache needs to coexist symbiotically: As an example, if one workload wants to use a large portion of the cache while another workload needs only a small portion of the cache, both workloads can execute concurrently while satisfying their needs.

In order to accommodate the needs of all of the cores in the multicore CPU, the LLC tries to reduce the potential for set contention and the associated invalidations.

First, the LLC is significantly larger (e.g., on the order of multiple megabytes, reaching tens of megabytes for large contemporary multicore CPUs) than the typical LLC capacities from the single-core era (e.g., the L2 caches of that era, then the LLC of the CPU, were only hundreds of kilobytes in size). Second, the LLC has notably higher associativity, with modern CPUs implementing 24-way and 32-way set-associative caches. While this significantly increases the area of the LLC, with some multicore CPUs using as much as half of their total die area for the LLC, the increases are amortized across all of the cores (as compared to increases in the private cache area).

Constructing Large Caches Modern LLCs typically make use of hierarchical designs to provide the large capacities required by contemporary workloads. A modern LLC can contain hundreds of thousands of cache sets, and if a CPU designer were to maintain a monolithic array for the entire LLC capacity, they would have to deal with very large wire delays, as a bitline would potentially be shared by hundreds of thousands of rows, increasing the capacitance and resistance of both the wire itself and the parasitic impacts of each attached row. The increased capacitance and resistance translate into linear increases in delay and energy and if left unaddressed can undermine the utility and efficiency of the LLC. Compounding the issue is the need for more cache *ports*, as multiple cores may want to access the cache at the same time, and a single-ported LLC can lead to starvation during periods of heavy cache request queuing.

To work around capacity and port limitations, modern caches rely on the concept of *cache slicing*, where a cache level is decomposed into *multiple* smaller pieces. (Cache slicing has been referred to by several terms over the years, including cache interleaving and cache multibanking. Early works on cache interleaving date to 1980 (Smith 1982) and initial implementations interleaved (i.e., distributed) the words within a single cache block across multiple cache banks. This chapter uses the term cache slicing to refer to cache block interleaving across cache slices/banks, where a single slice/bank contains the entire cache block (Sohi and Franklin 1991).) Each cache slice contains a subset of the total number of cache sets in the level and is further partitioned by the number of ways. The slicing requires each cache access to first go through a slice decoder to identify which slice to look up. While early slice decoder implementations used a subset of the cache block address bits (e.g., using two of the cache index bits to decide which of four slices the block is assigned to), several commercial CPUs employ complex hash-based set mapping functions (e.g., Lempel 2011) to avoid *hotspots* (i.e., the uneven distribution of cache accesses across slices).

Inside a slice, to further manage scalability, the slice is typically partitioned into a *tag store* (i.e., a tag directory, which stores metadata about each cache block in the slice) and a *data store* (i.e., the actual data inside each cache block). While tag–data partitioning is an independent concept from cache slicing, it also helps to reduce cache energy by not activating the data bitlines unless a cache hit occurs (albeit at the expense of increased latency due to serializing the tag and data lookups). Note

that for very large caches, the data store of a cache slice can be further partitioned into multiple two-dimensional *subarrays* of SRAM (Huang et al. 2013), to further reduce power consumption.

There are two general approaches to laying out the shared LLC on a multicore CPU chip, both of which support cache slicing. As mentioned in the section titled "Optimizing CPU Cores for Parallelism," a multicore CPU is made up of multiple tiles, where a tile includes a CPU core and its private caches. The first approach for LLC layout allocates a fixed amount of cache outside of these tiles, as shown in Fig. 8a, using a separate tile for the LLC. Often with this layout, the controller for the LLC is centralized, and each core has access to the cache controller via a *bus* (i.e., a shared wire across all cores), though some implementations maintain an independent controller for each cache slice, in which case a *crossbar* is used. The second approach for LLC layout includes a slice of the LLC with each CPU core tile, as shown in Fig. 8b (leading to a one-to-one correspondence between core count and cache slice count). With this layout, each cache slice typically maintains an independent controller, and the controllers are connected together using a *ring interconnect* (Lempel 2011; Huang et al. 2013).

A key challenge with large LLCs is access time. Conventionally, monolithic caches provide *uniform cache access*, where all cores can access any block in the cache with the same latency (assuming no interference). As LLCs have become larger and sliced, cores often deal with *non-uniform cache access* (NUCA) (Kim et al. 2002), where it can take a longer time for a core to access a remote slice (i.e., the additional time required to traverse the interconnect) than it does to access its local slice. A NUCA cache can incur higher latencies than a cache with uniform

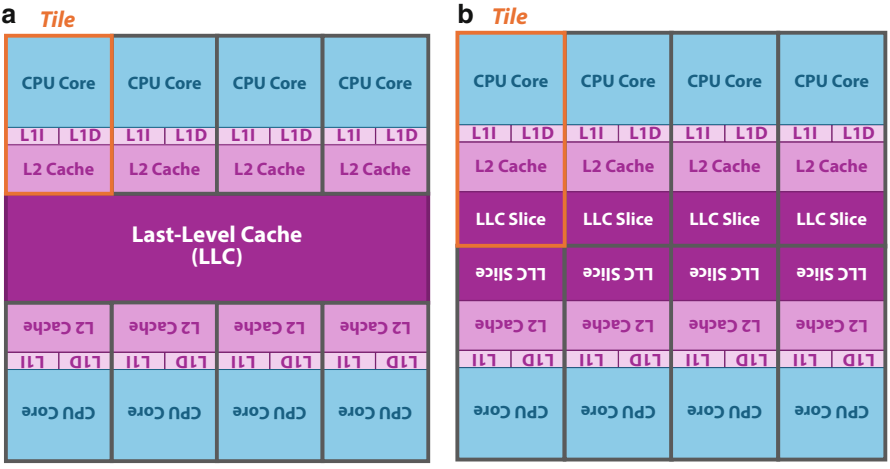


Fig. 8 Two approaches to tiling a multicore CPU layout. Note that tiles on the bottom row are intentionally shown upside down to represent the stamping of identical tiles. (**a**) Tile without a piece of the LLC. (**b**) Tile containing an LLC slice

cache accesses, but the slicing enables additional bandwidth by allowing concurrent accesses to each slice, mitigating some of the performance loss.

Main Memory Main memory modules (which consist of *dynamic random-access memory*, or DRAM, chips in modern systems) are connected to a CPU by a *memory channel* (see "Managing Memory" for details). Any core in a multicore CPU can access any of the CPU's memory channels, by accessing the on-chip *memory controller* that corresponds to the desired memory channel. As is the case for single-core CPUs, the memory controllers in a multicore CPU receive requests from the LLC. Given the sliced architecture of the LLC, each slice typically has the independent ability to dispatch cache misses to a memory controller. This is achieved by providing each slice with dedicated *miss status holding registers* (MSHRs) (Kroft et al. 1981), which make use of fully associative structures built using content-addressable memories (CAMs) to track in-flight cache requests. As there is not usually a one-to-one correspondence between cache slices and memory controllers (the typical multicore CPU has more cache slices than memory controllers), a slice can dispatch misses to any controller and chooses the correct controller based on how the system partitions memory addresses across memory channels.

For a given mix of workloads, a multicore CPU with n cores can potentially issue cache misses to main memory at as much as n times the rate that a single-core CPU does (note that this assumes similar core architectures in both CPUs and depends heavily on the specific workloads and on the impact of cache interference between cores). This requires the main memory to scale up its ability to respond to these cache misses, and modern memory subsystems have two examples of this. First, more MSHRs are provided at the LLC in order to sustain a greater number of concurrent memory accesses. An LLC miss must stall any time the cache does not have a free MSHR available to allocate. If there are not enough MSHRs in the system, then such stalls are more likely to occur. As each cache slice often has its own set of MSHRs, multicore CPUs take advantage of the increased slice count to provide more MSHRs at a low cost, compared to scaling up a monolithic CAM. Second, the memory bandwidth has been scaling up to accommodate the increased demand. The memory bandwidth provided by DRAM has been increasing significantly over the last two decades, through a combination of (1) increasing bus frequencies for the memory channel and (2) the emergence of new DRAM types such as DDR5 (JEDEC Solid State Technology Assn. 2024) and High-Bandwidth Memory (HBM) (JEDEC Solid State Technology Assn. 2020). While memory bandwidth can also be scaled by increasing the number of memory channels per CPU, multicore CPUs have largely avoided this approach due to the limited number of pins available in the CPU chip package.

The sharing of a single main memory across multiple cores can introduce new changes to the behavior of three properties, compared to their behavior with single-core CPUs. First, the optimal choice of an *address interleaving* scheme (i.e., which bits index which memory structures) can change depending on the type of workload

executing on the CPU. Second, the optimal choice of *row policy* (i.e., whether a DRAM row is left open after all currently queued requests are serviced) can depend on the type of workload as well. Third, the *memory scheduling algorithm* (i.e., the order in which queued DRAM requests are serviced) may need to change to avoid starving requests in certain scenarios. The section titled "Main Memory Policies" discusses each of these in detail.

Coordinating Memory Requests Across Cores

One key artifact of the memory hierarchy of a multicore CPU is that cores do not always write updates to globally visible locations. For example, in a typical multilevel hierarchy that employs write-back caches, a core will write data to its private L1 data cache, but this updated value will not be immediately visible to other cores in the CPU, as they cannot directly access another core's private cache. This is at odds with the architectural model (i.e., abstraction) that is presented to programmers, where the cores have access to a global *shared memory* (which is not the same as main memory; see "Shared-Memory Model" for more details). As part of this shared-memory model, a data update written by a core to shared memory will become visible to other cores. In contrast, caches are a *microarchitectural* optimization that is not, in principle, exposed to programmers as part of the architectural abstraction. (This is in part because (1) caches are designed to be hardware-managed structures that operate transparently to the programmer and (2) cache configurations can differ between different CPU models belonging to the same instruction set architecture. In practice, programmers use knowledge of the design of the cache hierarchy to optimize program performance, while still expecting the behavior of the system to adhere to the semantics of the shared-memory programming model.) As such, any interaction that a core has with a cache must *appear* to the program as if it is taking place globally in shared memory, as this is what a programmer expects.

In order to maintain program correctness, multicore CPU must ensure that data updates are coordinated across all of the cores (as is the case in any parallel computer architecture). For the shared-memory programming model typically used for multicore CPUs, this involves two types of coordination (Nagarajan et al. 2020): (1) *cache coherence*, which ensures that updates to a *single* unit of data (e.g., one cache block) are made visible to and ordered across all cores, and (2) *memory consistency*, which ensures that updates across *multiple* units of data are interleaved according to a predetermined policy across all cores. The sections titled "Cache Coherence" and "Memory Consistency Models" discuss cache coherence and memory consistency, respectively, in more detail.

Scaling to Many Cores

While the term *multicore* does not in theory place any limits on the number of cores in a CPU, there is a distinction made between multicore CPUs with smaller

core counts (e.g., under 24 cores in contemporary systems) and *manycore CPUs*, which are multicore CPUs that consist of several dozens of cores. This distinction is made because of key scalability challenges that become prominent as the core count increases significantly. For the smaller core counts, the interconnects described in the section titled "Sharing Caches and Main Memory" can enable reasonable parallel performance. However, at the manycore level, the significant increase in contention makes both bus-based and ring-based interconnects infeasible. As a result, specialized research and development has focused on how to provide more scalable communication at high core counts.

The initial ideas that would evolve into manycore CPU design stem from the Raw microprocessor project at MIT (Waingold et al. 1997). Conceived around the same time as the Hydra CMP (see "What to Do With All These Transistors?"), the Raw CPU took a more extreme approach to CPU core simplification, arguing that sophisticated compilers could offload the need for complex ILP mechanisms. As a result, the Raw CPU consisted of multiple small *tiles*, where each tile included a very simple CPU core along with a small piece of cache. The tiles were meant to be composable: Depending on the needs of a platform and on the available transistor count, a manufacturer could stamp out more or fewer tiles depending on their needs. By making each tile small, the distance between tiles (and, thus, the distance between cores) and intra-tile wire lengths would both be short, allowing for the CPU to run at a faster clock frequency without excessive power consumption. To enable composability with varying tile counts, the tiles connected to each other using a packetized *mesh network* (a two-dimensional interconnect), including configurable on-chip routers. The first Raw CPU, prototyped in 2002, included 16 tiles on a single chip.

Unfortunately, there is no precise definition of a core count or of specific properties that a manycore CPU must have, and the perceived distinction between manycore CPUs and more conventional multicore CPUs has shifted as capabilities have evolved over the last two decades. Aside from mesh-based tile organizations, other properties observed in several manycore CPUs include non-uniform cache access (NUCA) architectures with multiple cache slices and the replacement of hardware cache coherence with software-driven message passing interfaces. While not a comprehensive list, examples of commercial CPUs that have been considered to be manycore include the Tilera TILE 64 (a commercialization of the MIT Raw CPU), the Intel Xeon Phi series, and various CPUs from PEZY, including their 2048-core PEZY-SC2 released in 2017.

Managing Memory

The cores in a multicore CPU share memory with each other, as discussed in the section titled "Sharing Caches and Main Memory." This sharing introduces a number of issues that must be considered by both programmers and architects. This section will discuss several of these issues. To start, it defines what it means to share memory, both from a software perspective and from a hardware perspective. Then,

it discusses main memory management and how management policies in hardware can impact the performance of threads running on a multicore CPU. Finally, it discusses examples of cache coherence and memory consistency protocols that ensure correct program behavior in a shared-memory environment.

To guide the discussion in this section, Fig. 9 shows an example of how the hardware in a memory subsystem is shared across the cores in a multicore CPU. All of the cores in a multicore CPU share a last-level cache (LLC; see "Sharing Caches and Main Memory" for LLC design details). If a cache request misses in the LLC, it must go to main memory, which is made up of DRAM. The physical address space of a system enables each byte physically available within the main memory to be accessed using a unique address. The physical memory is split up across one or more *memory channels*, where each channel connects to one or more DRAM *modules*. Each channel is managed independently of one another, and access management and maintenance tasks (e.g., refresh) are handled by a dedicated *memory controller* for the channel (see "Main Memory Policies"). Each module contains a series of DRAM chips, which are grouped into one or more *ranks*. All the chips belonging to a rank operate in lockstep (i.e., they always perform the same operations on the same row/column of the same arrays in each chip). A rank consists of several *banks* of memory, with each bank physically striped across all of the chips in a rank. Logically, a bank operates as a single two-dimensional array of DRAM cells, where a cell consists of a capacitor and a transistor and can hold one bit of data. Due to the small charge capacity of the capacitor, the memory controller cannot directly perform data operations on the DRAM cell and instead loads (i.e., activates) one row from a bank at a time into the *row buffer*, from which the controller can issue reads from and writes to the activated row.

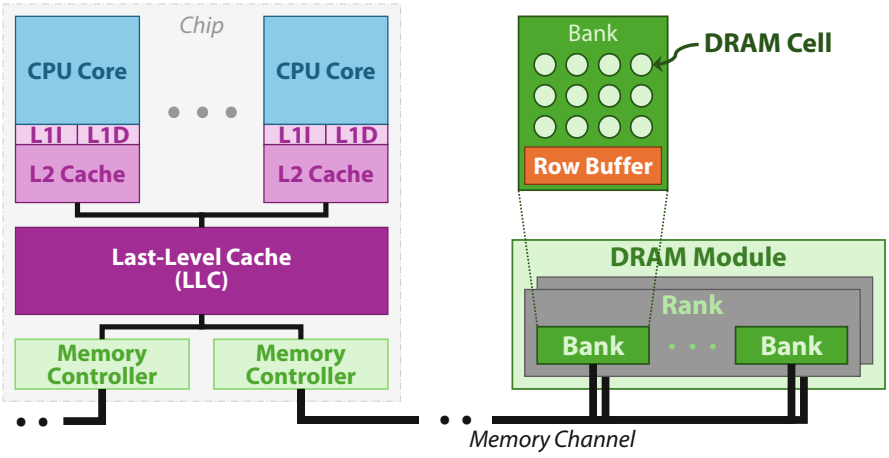


Fig. 9 Example memory hierarchy for a multicore CPU, showing shared on-chip last-level cache and shared off-chip main memory

Shared-Memory Model

From the first symmetric multiprocessor computer, the Burroughs D825 (Anderson et al. 1962), the majority of multiprocessor systems have enabled CPU cores to share the physical main memory with each other. Through the decades, to mitigate the long latencies of memory, several microarchitectural enhancements (e.g., caches, memory speculation) have extended the memory subsystem hardware but have done so transparently to the programmer. From the programmer's perspective, multicore CPUs predominantly make use of a *shared-memory model*, which at a high level resembles primitive computers. In the shared-memory model, each core has access to a single shared physical memory (which may be implemented in a centralized or a distributed manner). When any core performs a store to an address in this single shared memory, the update becomes visible to all cores that subsequently load from that address. While in principle the shared-memory model sounds simple, it has two key implications on the design of multiprocessor systems (and, by extension, multicore CPUs).

First, for a core, one needs to define when a store is considered to be performed (i.e., the moment at which the stored value becomes visible to other cores). For example, for an out-of-order core, a popular definition is that a store is performed once the store instruction is committed. Prior to commit, there are several events that can cause an executed store to be squashed (e.g., exception, misspeculation), making the commit event the earliest time that the store is guaranteed to safely proceed. As a result, stores are buffered inside the core, and the store operation to the first-level cache is not initiated until after the store instruction commits. A physical *store buffer* (sometimes called a *write buffer*) contains all committed writes that are yet to be issued to the cache. Depending on the memory consistency model, this store buffer may or may not be considered a part of shared memory; if it is, the store values in the buffer must be made available to other cores in a way that does not violate the consistency model (see "Memory Consistency Models" for more).

Second, the CPU must ensure that microarchitectural optimizations for memory operations obey a consistent set of rules, which are part of the instruction set architecture (ISA), so that programmers do not encounter program correctness issues during execution. All private and shared caches are considered part of the shared memory, which means that cache coherence is needed to propagate the value of a cache store to other caches and cores, as the section titled "Cache Coherence" discusses. Both within a thread and across threads, the interleaving of a store with other stores and with loads must obey a set of rules that are promised to the programmer, which are defined as part of the memory consistency model. As "Memory Consistency Models" discusses, some memory consistency models have strict interleaving expectations, which can simplify programming complexity at the expense of high performance overheads, while more relaxed memory consistency models can allow different cores to observe different interleavings (following a defined set of guarantees) in order to improve performance.

Main Memory Policies

As discussed in "Sharing Caches and Main Memory," there are three types of policies for main memory management that are impacted by the introduction of multiple cores: address interleaving, row policy, and memory scheduling. Each of these policies is implemented inside the memory controller. Unlike the single-core case, where the choice of policy for each type can be determined by analyzing program behavior, the nondeterminism that exists for shared-memory interactions in a multicore CPU makes it difficult to easily choose a single optimal policy.

Address Interleaving Address interleaving (also known as memory interleaving) schemes (Balasubramonian 2019) determine the way in which address bits are assigned to index different levels of the main memory organization, in an attempt to take advantage of spatial locality and memory-level parallelism (MLP). As shown in Fig. 9, a typical DRAM-based main memory includes one or more memory channels, where each channel contains one or more DRAM modules. Within each module, there are multiple chips, with the chips grouped into one or more ranks. Each rank (a group of DRAM chips that operate in lockstep) contains multiple banks of DRAM, where each bank can mostly operate in parallel with each other. Inside each bank, at most one row of DRAM can be open at a time, where a row holds multiple columns of data (with each row in commonplace DRAM types containing a few kilobytes of data).

Figure 10 illustrates two common interleaving schemes that optimize locality and MLP over a basic approach without interleaving. In the first scheme, *cache block interleaving*, consecutive cache blocks in the memory address space are distributed to different banks (and in some cases to different memory channels) to allow requests to the two blocks to be serviced concurrently. In the second scheme, *row interleaving*, consecutive cache blocks stay in the same row to maximize hits to the already-open row, but consecutive rows are distributed to different banks/channels. While cache block interleaving is a popular scheme for single-core CPUs, it can introduce issues in a multicore CPU. For example, with a multiprogrammed workload (i.e., when multiple cores are executing threads that belong to different processes), one thread may tie up many channels at the same time by accessing several consecutive cache blocks. This would increase the chance of interference for any of the other threads trying to access data in memory, by increasing the likelihood that the channel will be busy. Worse off, if two different processes access several consecutive cache blocks, this can generate a large number of *bank conflicts* (i.e., *row conflicts*) in DRAM that could have been avoided with a row interleaving scheme. A non-interleaved memory (i.e., where consecutive rows map to the same bank) can reduce the probability of bank conflicts due to interference, but at the expense of sacrificing memory-level parallelism for memory-intensive applications.

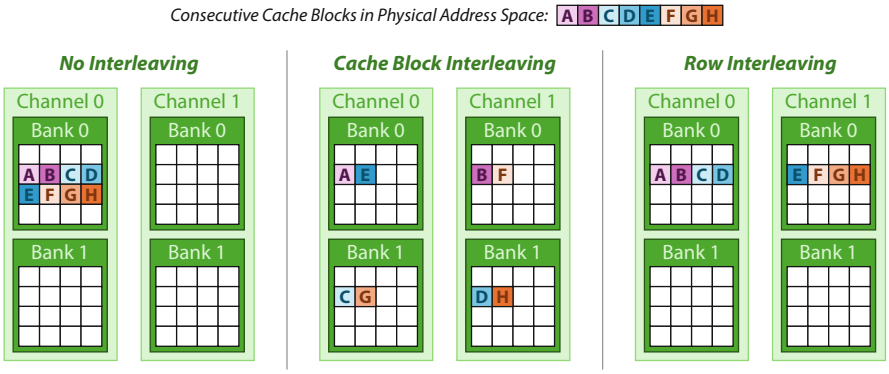


Fig. 10 Example of how address interleaving schemes impact the placement of cache blocks in main memory, for a series of cache blocks that are consecutive in the physical address space. A specific interleaving scheme is chosen by determining which bits of the physical memory address are used to index the different levels of the memory hierarchy (in this example, the channel, the bank, the row, and the column)

Row Policy The row policy (Balasubramonian 2019) determines whether a row in a bank should remain open once the memory controller services all currently queued requests for that bank. In an open-row policy, the row stays open in case future memory requests also access the same row (due to spatial locality), avoiding the latency of reopening (i.e., reactivating) the row when the next request to the bank arrives. In a closed-row policy, the row is closed as soon as the currently queued requests for that bank are completed, under the assumption that new requests are likely to access a different row, thus avoiding the row closing (i.e., precharging) latency when the next request to the bank arrives. Variations of these policies exist, such as timeout policies that close the row a given number of cycles after the last queued request is serviced. In a multicore CPU, the optimal policy choice can depend on the workload executing on the CPU. For example, with a multiprogrammed workload, the likelihood of the next access to the bank being to the same row decreases compared to a single-core CPU, due to interference and competition across different processes. However, with a multiprogrammed workload, the likelihood can potentially increase instead, as multiple threads belonging to the same process may each access data in the same row, depending on how data accesses are partitioned across threads.

Memory Scheduling The scheduling algorithm (Balasubramonian 2019) determines which of the pending memory requests to service next, which impacts the rows and banks that are currently active in DRAM. As mentioned above, a DRAM read or write cannot be directly performed on the DRAM cells and can only perform the reads or writes on data in the row buffer of a bank (which

practically limits a bank to servicing at most one request at a time). This results in significant complexity for scheduling memory requests, as the controller must consider many factors such as the time a request has been waiting for, whether the target bank for each queued request is idle or is in the middle of servicing another request, and whether the target bank has the target row currently open (i.e., activated). Furthermore, while a memory controller can take advantage of *bank-level parallelism* (BLP) to service requests to *different* banks concurrently, the physical bus wires of the memory channel are shared across all banks belonging to the channel, as shown in Fig. 9. As a result, the scheduler must also stagger requests in a way that ensures exclusive access to the memory channel bus for only one bank, when the bank needs to send data to or receive data from the controller. Figure 11 shows how this staggering of memory requests can be coordinated for two examples: (1) when multiple requests target different rows in *the same bank* (i.e., a *bank conflict*) and must serialize the opening of each row and (2) when multiple requests target different rows in *different banks*, which can exploit BLP to overlap multiple row openings and must serialize only the actual data transfer on the shared memory channel's data bus.

Ultimately, a scheduling algorithm enforces these many constraints by prioritizing currently queued requests based on a predefined policy and on the current state of the DRAM and by enforcing a series of *timing parameters* that are declared as part of the specification of each DRAM type (e.g., DDR5 (JEDEC Solid State

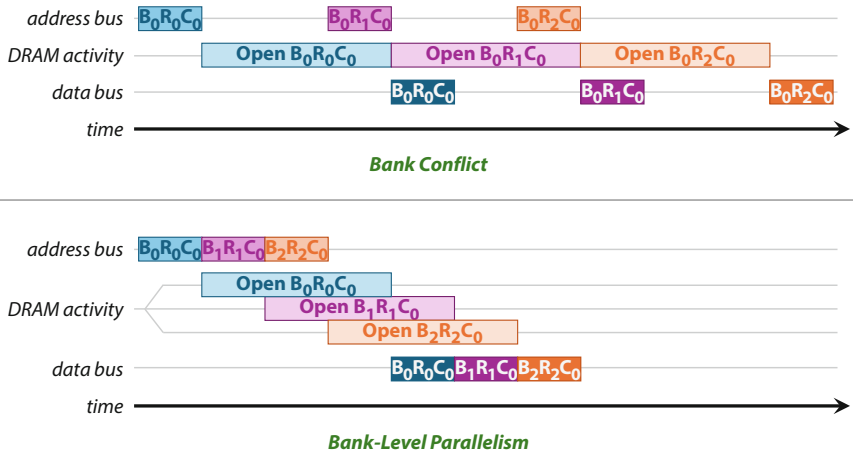


Fig. 11 Scheduling of multiple read requests when a bank conflict occurs (top) and when there is bank-level parallelism that avoids a bank conflict (bottom). The notation $B_b R_r C_c$ indicates a memory access to Bank b, Row r, and Column c. This example assumes that all requests access the same memory channel. The memory controller for the channel sends a request to the memory via the address bus just before it is safe to initiate the operation inside memory (e.g., opening a row). The controller must ensure that operations to the same bank do not overlap *and* data transfers from memory to the controller on the shared data bus do not overlap

Technology Assn. 2024) and HBM (JEDEC Solid State Technology Assn. 2020)). A commonly implemented algorithm is known as first-ready, first-come first-serve (FR-FCFS) (Rixner et al. 2000), which prioritizes requests to already-activated rows over older requests. FR-FCFS does this to improve *average memory access time* (AMAT), as the time required to activate a row, and to precharge (i.e., close) the row after requests are finished, can both take as long as the read and write requests themselves (JEDEC Solid State Technology Assn. 2024). Memory controllers in commercial multicore CPUs continue to use FR-FCFS without introducing any notion of thread or application awareness (Balasubramonian 2019), which can exacerbate interference across threads (see "Mitigating Interference").

Mitigating Interference

The combination of private caches and shared caches attempts to balance the impact of interference with the need to provision resources for worst-case behavior: Private caches allow each CPU core to manage a small working set of its data without first-order impacts of interference, while the shared cache enables resource pooling to reduce the frequency of long-latency misses to main memory for workloads with large data footprints. However, it is still possible for memory interference, at the caches and/or at main memory, to negatively impact one or more cores. This section looks at three such examples, as well as potential mechanisms to mitigate them.

Our first example examines how cache requests from one core can impact the management of a private cache belonging to a different core. If the shared LLC is inclusive of upper-level private caches and one core evicts a cache block from the LLC belonging to a second core, the second core will also have to evict that cache block from its private caches *even though the private cache space is not being used by the first core*. To address this (as well as pollution from hardware prefetchers), some multicore CPUs employ a *noninclusive* or *exclusive* policy for the LLC or for the last private cache level (Backes and Jiménez 2019). On a miss to main memory, the noninclusive policy acts like the inclusive policy, where a copy of the data is placed in the LLC and in the private cache(s). When a cache block eviction takes place at the LLC, the noninclusive policy acts like the exclusive policy, where the eviction does *not* trigger an eviction in the upper-level private caches.

Our second example examines how shared cache utilization by one core can impact the available shared cache capacity, and thus the performance, for a different core. In a conventional multicore CPU, shared resources are left *unmanaged*, i.e., there is no active mechanism to enforce that each core receives a fair portion of the resources. This unmanaged approach is often useful to allow the easy distribution of these resources across cores based on the heterogeneous needs of the threads that each core is executing (e.g., in a two-core CPU, one core runs a thread that has a large *working set* and uses up most of the LLC, while the other core runs a thread that has a small working set and can make do with whatever capacity the first thread does not use). However, when the heterogeneous needs are more extremely unbalanced, one or more of the threads may incur significant *slowdowns*. In our

example, if one greedy thread is using most of the LLC, cache blocks belonging to other threads may constantly be evicted, hurting those threads' performance significantly. To address this, several works have proposed *cache partitioning*, where some or all of the ways and/or sets of a cache are assigned to a specific core or thread. Strict cache partitioning could ensure that the cache blocks from the other threads in our example do not get evicted, as the greedy thread would not be allowed to use partitions belonging to other threads. One commercial implementation of cache partitioning is Intel's Cache Allocation Technology (Herdrich et al. 2016).

Our third example examines how memory scheduling algorithms can introduce unintentional slowdown for threads. Recall from "Main Memory Policies" that the commonly used FR-FCFS scheduling algorithm does not account for any sort of thread or application awareness and solely focuses on reducing the number of row activations and precharge operations. While reducing activate and precharge operations help decrease the AMAT for a thread that can exploit row locality, they can unfairly slow down the other threads in a multicore CPU, by continuing to prioritize requests from the thread whose row is already activated. In this example, one thread is generating many loads and stores to the same DRAM row, which is currently activated, while the other threads each have a single load that is waiting to access a different row in the same bank. FR-FCFS will look at all of the queued requests and prioritize the requests from the first thread because their target row is already activated. This causes all of the requests from the other threads to wait longer in the queue and in extreme cases can lead to unintentional starvation for these threads. To address this, researchers have proposed a number of memory schedulers that explicitly augment the memory request metadata with a thread ID and incorporate information about the thread into the scheduling algorithms. As one example, the memory controller can use lightweight runtime metrics to predict which threads are being slowed down due to memory interference and can prioritize requests from these threads before prioritizing row locality (Mutlu and Moscibroda 2006).

Cache Coherence

As mentioned in "Sharing Caches and Main Memory," each core in a multicore CPU has private caches in addition to a shared cache. Similar to conventional single-core CPUs, these caches are typically configured to be *write-back caches*. A write-back cache buffers updates to a cache block, by storing the updates in the cache and marking the cache block as *dirty*. In a single-core CPU, these updates are not made visible to lower cache/memory levels until the cache block is evicted from the current cache level. Upon eviction of a cache block, if the block is marked as dirty, its contents are written to the next lower level to preserve the most recent version of the data; otherwise, the block can simply be dropped, as at least one lower level of the memory hierarchy also has the most recent version of the data. For cache blocks that are written to frequently, write-back caches reduce the amount of traffic these writes induce on lower levels of the cache hierarchy. Write-through caches,

a common alternative, store the update in the cache *and* send every update to the next lower level. However, write-through caches are not commonly employed in multicore CPUs due to the impact of the increased write traffic on interference.

The Need for Coherency with Write-Back Caches In a single-core CPU, write-back caches ensure that the CPU core sees the most recent version of the data for each cache block, as the core always starts memory lookups from the top cache level in the memory hierarchy. However, with multicore CPUs, a dirty cache block stored in one core's private cache is not immediately visible to other cores in the CPU. This is because if the same cache lookup procedure for single-core CPUs is used, a multicore CPU's core would access its own private caches and the shared last-level cache and *not* the private caches of other cores. If left unmodified, this can lead to correctness issues, as one core may ignore and potentially incorrectly overwrite the updates from another core.

To address the lack of visibility, multicore CPUs employ hardware mechanisms to support cache coherence. Cache coherence techniques were initially developed for parallel systems with multiple single-core CPUs, but the same basic techniques have become applied for multicore CPUs. At a high level, a cache coherence mechanism ensures the following rules: (1) A write operation to a memory address by a processor must (eventually) become visible to *all* processors, and (2) if two processors write to the same memory address, *all* processors will observe the writes in the same order. This enforcement is typically achieved by guaranteeing two properties: (1) At any given time, at most one core is granted permissions to write to a memory address by the coherence mechanism (note that these are different than the write permissions maintained by virtual memory), and (2) when a core has been granted write permissions to a memory address, any copies of that address held in other private caches are marked as stale and cannot be used again without first synchronizing data updates from the writing core. There are multiple *cache coherence protocols* that enforce these properties in different ways, and a multicore CPU will typically implement one of these protocols as a fixed-function mechanism in hardware. Note that cache coherence mechanisms are fully *microarchitectural*: As caches are part of the microarchitecture and are (in theory) transparent to the programmer, the programmer expects the machine to obey shared-memory model semantics, and a value written to a specific memory address should be visible to *all cores* as soon as the write occurs.

Logically, a cache coherence protocol maintains a *coherence state* for each core for every memory address. The coherence state dictates the current permissions that the protocol has granted a core for a particular memory address. As this would require a large amount of data to be stored on chip, practical implementations of cache coherence make the following two optimizations. First, coherence states are maintained at a *cache block granularity* and not per byte (i.e., per memory address). One artifact of block-based coherence management is *false sharing* (Torrellas et al. 1990). If two CPU cores are accessing two different memory addresses that belong to the same cache block, the cache coherence protocol treats this as data that is

shared by both cores, even though their accesses are to different pieces of data. In the case where both cores are trying to write to these pieces of data simultaneously, the cache block will ping-pong (i.e., move back and forth) between the two cores, with one core invoking an invalidation for the other core any time it wants to write and vice versa. False sharing can be avoided only by ensuring that the two pieces of data are mapped to different cache blocks (e.g., by having the programmer pad data structure sizes to use up exact multiples of the cache block size). Second, if a cache block is not currently held by a core in its private caches, the cache block is assumed to be in a coherence state designated as *invalid* for that core (i.e., the core cannot currently read from or write to that cache block).

Exchanging Coherence Messages Between Cores The cache coherence protocol is typically triggered when a core wants to change its coherence state for a cache block, which can happen for one of the three following actions: (1) the core accesses a cache block that it does not currently have in its private caches, (2) the core wants to change its current permissions under the coherence mechanism for the cache block, or (3) the core evicts the cache block from its private caches. If the rules of the protocol require a state change for the action, this generates a coherence message. The specific sequencing of messages is protocol-dependent but typically involves sending state upgrade/downgrade requests and receiving acknowledgments and potentially updated data values. There are two approaches to sending messages to a destination core: snoopy coherence and directory-based coherence.

In *snoopy* cache coherence (Goodman 1983), cores share a bus, and each core maintains its own coherence state metadata. Each core reads *all* cache coherence messages transmitted on the bus (i.e., it is snooping on all messages) to see if it needs to react to the message (e.g., if it needs to write back any dirty changes to a cache block and if it needs to invalidate the block). For example, if a core wants to write to cache block x, it broadcasts a coherence message on the bus declaring its intent to acquire write permissions. As only one core can have write permissions, every other core on the bus will see the coherence message, and if a core has a copy of cache block x, it will invalidate the block in all of its private caches. The other cores require some form of acknowledgment mechanism to notify the requesting core that it has completed any necessary actions or if it has an up-to-date version of the data. Snoopy cache coherence is the simpler of the two approaches to implement and works effectively when the CPU has a relatively low number of CPU cores, but the broadcast-based bus scales poorly as the core count increases, both due to the increased bus latency/energy and increased contention due to message serialization on the bus.

Directory-based cache coherence (Censier and Feautrier 1978) avoids the poor scaling of broadcasting by instead storing the coherence state metadata in a directory. When a core triggers a coherence message, the message first goes to the directory, which stores the state of the cache block in all cores serviced by the directory. One example implementation is to maintain a bitvector for each cache block currently held by *any* of the cores, where bit i in the bitvector indicates

whether core i currently has a copy of the cache block. When the directory receives a coherence message, it looks up the metadata for the requested block and then dispatches follow-up messages to only those cores that currently hold the cache block. While directory-based coherence is more complex to implement, it exhibits scalability over snoopy coherence in multiple dimensions. First, as mentioned above, coherence messages need to be transmitted to only the cores that currently hold the data and can be implemented using more scalable interconnection networks than a bus. Second, the directory can be partitioned into multiple slices, with each slice responsible for a subset of memory addresses. Third, a CPU can implement multiple levels of directories, creating a directory hierarchy that further reduces coherence message traffic and metadata storage requirements. This improved scalability makes directory-based coherence a good fit when there are a large number of cores in the CPU.

Coherence Protocol Examples As mentioned above, there are many specific instances of cache coherence protocols. Figure 12 illustrates the state transition diagram of MSI (Censier and Feautrier 1978), a popular coherence protocol. MSI takes its name from the three possible states a cache block can be in: (1) *modified*, which gives a core permissions to read from *and* write to the cache block, (2) *shared*, which gives a core permissions to read from the cache block, and (3) *invalid*, which means that the cache block is not in the cache (and the core that the cache corresponds to therefore cannot read from or write to the block). For the sake of simplicity, let us examine these states for a system with four CPU cores, where each core has its own private L1 cache, and the cores all share a single L2 cache. When a

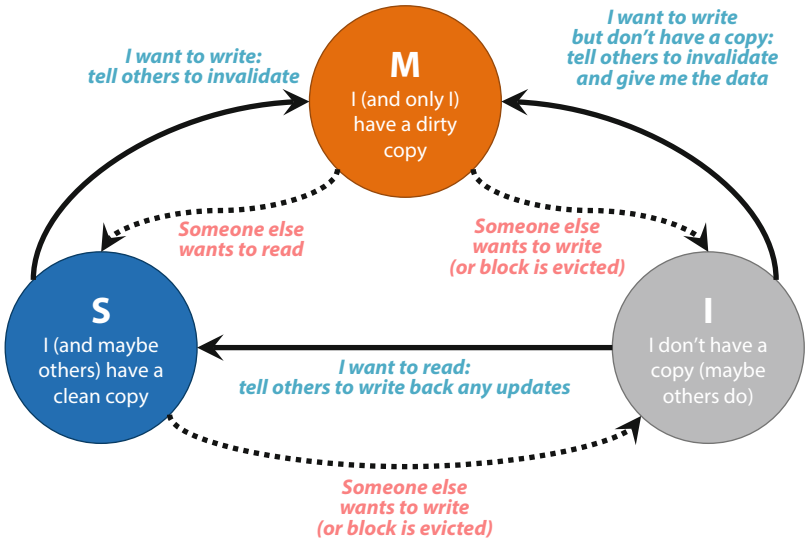


Fig. 12 MSI protocol: solid lines are upgrades and dotted lines are downgrades

cache block is in the shared state, multiple L1 caches can hold identical copies of the cache block. Since none of the cores have permissions to write to the cache block, this ensures that any core reading the cache block has the most recent version of the data. When a thread running on one of the cores wants to write to the cache block, it first sends a message to the other cores, informing them to downgrade to the invalid state (i.e., to invalidate their copy of the cache block, if they have one). Once the core receives acknowledgments of the invalidations, it upgrades its own copy of the cache block to the modified state. (While not shown, practical cache coherence protocols implement additional *transient states*, to indicate an in-progress upgrade/downgrade while waiting for other cores to complete their requested state changes.) This now ensures that only this core has a copy of the cache block and that the core can now safely perform its write. If another core wants to read the now-modified cache block, it will send out a read request, which will force the writing core to downgrade its cache block from the modified state to the shared state and to make the updates visible to the other cores.

One drawback of MSI is that even when only one core has a copy of the cache block in the shared state, it must wait for acknowledgments from all of the other cores before it can upgrade this cache block to the modified state. This can potentially generate a large amount of unnecessary coherence traffic, given the lack of other copies in the private caches. The MESI cache coherence protocol (Papamarcos and Patel 1984) provides an optimization for this drawback, by introducing a fourth state called *exclusive*, as shown in Fig. 13. If a core reads a cache block into its private cache and no other private cache has a copy, the cache block will have an exclusive state, indicating that the core can read from the block *and* that no other private copy exists. If the core subsequently wants to perform a write to the block, it can now silently upgrade the cache block (i.e., without sending any coherence messages to other cores) to the modified state. To ensure correctness, if a second core wants to read the cache block, the first core's copy is downgraded from exclusive to shared, indicating that there may be more than one core currently holding a copy of the block.

Memory Consistency Models

A key consideration in coordinating memory updates across cores is the need to present a globally consistent view of these updates to all cores. While cache coherence ensures that updates made by any one core to one unit of data are visible immediately to all cores, memory consistency (Nagarajan et al. 2020) deals with how requests are interleaved across multiple units of data. Unlike cache coherence, which is a purely *microarchitectural* technique (i.e., transparent to the programmer) because it is a result of only microarchitectural design choices, memory consistency is an *architectural* technique that must be exposed to the programmer, because data communication across threads is determined by the programmer in both implicit and explicit ways. There are many different memory consistency models, where

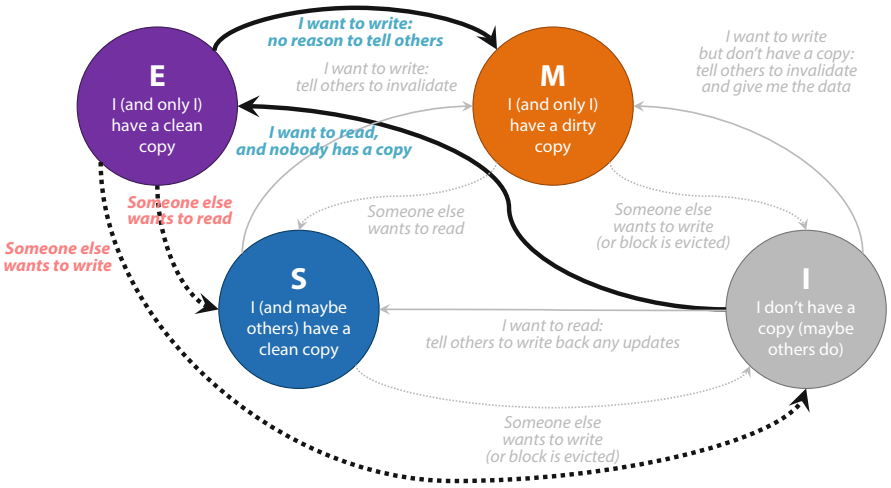


Fig. 13 MESI protocol: solid lines are upgrades and dotted lines are downgrades. Transitions that are unchanged from MSI are grayed out

each memory consistency model defines which possible memory interleavings can be observed by a core. Regardless of the consistency model, each core must maintain an ordering of loads and stores that does not violate true (i.e., read after write) register dependencies, and because caches are part of the shared-memory state, cache coherence is implemented. This section examines three popular types of consistency models.

Sequential consistency (SC) (Lamport 1979) ensures that *every core* sees the same ordering (i.e., interleaving) of individual memory operations. This is the equivalent of cores going one at a time when performing a load or store, and that load or store is considered part of the shared-memory state so that all cores see the effect (though not all cores need to see the effect immediately). SC is often thought of as one of the most intuitive models, but given its need for a globally consistent interleaving, it can require a high overhead for implementation and is rarely used for modern multicore CPUs.

Relaxed memory consistency models allow for some differences in orderings observed by each core. One example of a relaxed model is *total store ordering* (TSO) (SPARC International Inc. 1991), which was first developed for the SPARC ISA and is used widely (albeit with some modifications) by the x86 ISA. In TSO, a CPU core can observe its own write before other cores can, resulting in a slightly different interleaving observed by each core. A key goal of TSO is to retain *store buffers* in multiprocessors. For an out-of-order core, a store buffer holds stores that have been committed but have yet to be written to the caches. In SC, the store buffer is not considered part of the shared-memory state, and memory speculation techniques that read data from the store buffer can require costly mechanisms to squash and replay out-of-order load that violate the global interleaving. With TSO's

relaxation, there is no need for such costly mechanisms, and the write buffer is considered part of the shared-memory state.

An example of an even more relaxed model is *weak ordering* (Dubois et al. 1986). Weak ordering allows most memory operations to be reordered but uses programmer-invoked synchronization primitives to explicitly define reordering boundaries. A popular synchronization primitive for weak ordering is a *fence*, which is used to enforce orderings within a core (but does not explicitly synchronize across cores, unlike a barrier). A fence has three guarantees: (1) All cores see the same exact ordering of fence primitives, (2) all load and store instructions that come before the fence in a thread must complete before the fence, and (3) no load or store instruction that comes after the fence can complete until after fence takes place. What this means is that for the loads and stores in between two fences, any ordering of them can possibly occur. The programmer can explicitly insert more fences into a thread to enforce a stricter ordering. The Arm ISA is an example architecture that makes use of weak ordering.

Please refer to a detailed discussion (Nagarajan et al. 2020) for more information about these and other memory consistency models.

Optimizing Operating Systems for Multicore CPUs

As mentioned in "Motivating the Need for Concurrent Processing," multicore CPUs can make use of two types of parallelism: multiprocessing of multiple applications and TLP within an application. In order to best exploit these types of parallelism on multicore CPU hardware, operating systems and user applications have evolved in a number of key ways. While these changes are not required to make use of multicore CPUs, they can allow developers to significantly improve the overall performance and efficiency of the system. This section briefly touches on operating system changes and optimizations for multicore CPUs.

To facilitate our explanations, these software optimizations will be explained from the perspective of *threads*. As a simple broad definition, a thread is a sequence of CPU instructions from a program and serves as the basic unit of scheduling for the operating system. Generally speaking, when a program starts executing, that specific executing instance is known as a *process*, and a process can consist of one or more threads of execution.

Scheduling Threads One of the many tasks that an OS is responsible for is scheduling threads from all currently running processes. For a single-core CPU without multithreading, the CPU can execute only one thread at any given time. As discussed in "Multiprocessing," the OS maintains the illusion of concurrent thread execution by time-sharing the CPU. At the end of a scheduling quantum, the OS uses a predefined *scheduling policy* to select which of the active threads will execute on the CPU for the next quantum. For the sake of simplicity, let us assume for now that the OS selected a new thread that is just starting its execution. The selected thread is then assigned to the CPU's *hardware context*, and all of the CPU's *states*

for that thread (e.g., program counter, registers, predictor history) are initialized to the starting state. The thread then executes until the end of the quantum, unless the thread is preempted early (e.g., to execute an exception handler).

Assuming the typical conditions that no early preemption took place and that the thread has not yet finished executing, at the end of the quantum, the OS invokes the thread scheduler to select the thread for the next quantum. If the selected thread is different from the currently running thread, the OS invokes a preemption of the running thread, which is known as a *context switch*. During a context switch, the OS copies the CPU state associated with the thread being preempted into memory and loads in the CPU state for the thread being run next from memory. From the perspective of the thread, the context switch gives the thread the illusion that it never stopped executing. From the perspective of a user, PCs can often give them the illusion that all of their threads are running concurrently on a single-core CPU, by rapidly context switching between them, as the scheduling quantum is on the order of milliseconds and is imperceptible to humans for the typical number of concurrently running threads.

To extend thread scheduling for multicore CPUs, the individual cores are each exposed to the OS. As an n-core CPU without multithreading has n hardware contexts, the OS can schedule n threads for every quantum. (When a CPU supports m-way multithreading, each way is typically exposed to the OS as its own hardware context, meaning that an n-core CPU with m-way multithreading exposes $n \times m$ hardware contexts.). Conventionally, OS schedulers treated hardware contexts as identical to one another, but this introduces two challenges in modern multicore CPUs.

First, while a context switch preserves and restores CPU state for a thread, this state does not typically include the contents of the cache, because cache blocks are meant to be quickly accessible copies of data that is available in other parts of the system. However, it can be beneficial for a thread to be reassigned to a hardware context that it was previously scheduled on, as the thread can take advantage of data that it had previously cached. Conventional scheduling approaches would disregard this and assign the thread to *any* available context. *Processor affinity*, sometimes referred to as *CPU pinning*, overcomes this by allowing a user to assign a thread for execution on only the user-specified hardware contexts (e.g., the user assigns a thread to only one CPU core). The OS scheduler obeys the processor affinity assignments, ensuring that the thread executes only on the selected contexts.

Second, as discussed further in "Heterogeneous CPU Cores," modern multicore CPUs no longer have homogeneous cores. As a result, the choice of hardware context to assign to a thread can have a significant impact on its performance and energy consumption (e.g., assigning a thread to a big core when it needs only a little core). Early systems with two types of cores made the difference between cores transparent to the OS: A single hardware context was associated with both a big core and a little core, and the hardware would use the CPU frequency setting chosen by the OS for the CPU (see *Controlling CPU Core Frequency* below) to decide whether the thread should run on the big core or on the little core. Modern systems can have

more than two types of CPU cores and often expose all of the CPU cores directly to the OS. To manage these cores efficiently, OSes now include heterogeneity-aware schedulers, such as the Energy Aware Scheduling approach available in modern versions of the Linux kernel (Linux Kernel Organization 2023b).

Controlling CPU Core Frequency As introduced in the section titled "Optimizing CPU Cores for Parallelism," multicore CPUs expose the ability to perform per-core DVFS. In modern systems, the voltage and frequency setting for each core is chosen by the OS. CPU manufacturers expose the DVFS capabilities as a series of frequency steps, where the OS chooses a target frequency based on certain properties, and the CPU uses the selected frequency step to control the voltage and frequency of the core.

While specific implementations vary, this section will focus on the Linux kernel implementation due to its readily available documentation. Linux includes a series of *governors* (Wysocki 2020), which are policies that the OS uses to select a frequency setting for a CPU. Some governors, such as *performance* or *powersave*, constantly run a CPU at a fixed frequency (for the two examples, maximum or minimum, respectively). Other governors, such as *ondemand* and *conservative*, use basic thread statistics such as load and idle time to select the frequency. Newer governors, such as *schedutil*, extract information about which threads were scheduled for the upcoming scheduling quantum to determine the frequency. In addition to these built-in governors, developers can create custom governor policies for their systems. A recent addition to the Linux kernel, Capacity Aware Scheduling (Linux Kernel Organization 2023a), combines core heterogeneity information with frequency settings chosen by the governor to guide hardware context assignment during thread scheduling.

Parallelizing Programs On the application side, there are two ways to take advantage of the parallelism and efficiency offered by multicore CPUs. The first method is to rewrite programs as *multithreaded applications*. Instead of writing a program as a fully sequential series of functions, a programmer can identify opportunities to perform some parts of the application concurrently. The programmer can use threading libraries to either (1) explicitly spawn these parallel parts into independent threads or (2) demarcate regions of the program that inform an advanced library to automatically generate threads. Note that threads do not have to be fully independent and that programmers can use *synchronization primitives* to coordinate execution across threads (e.g., locks to protect critical section execution, barriers to synchronize task or sub-task completion), as well as *shared memory* or *message passing* to exchange information between threads. While this chapter does not go into multithreaded programming in detail, there are other works (Mattson et al. 2004; Farooqi et al. 2022) that provide in-depth coverage of parallel programming techniques and frameworks.

The second method is often known as *multiprogrammed execution*, where multiple independent processes execute concurrently on the CPU. With multipro-

grammed execution, it is possible to use all of the cores in a multicore CPU even if all of the applications are single-threaded, by allowing the processes to execute in parallel. Most operating systems enable users to launch multiple processes concurrently, and when a process starts (initially with one thread), the OS adds the process' thread to the list of all active threads for the OS to schedule (see *Scheduling Threads* above). If a process is multithreaded, it will spawn additional threads over time, which the OS will also add to its list of active threads. The OS scheduler will typically not distinguish between threads belonging to the same process and threads from other processes and will schedule as many threads as there are hardware contexts.

Evaluating Multicore CPUs

While conventional metrics such as speedup have been widely used in the architecture community for decades, several challenges make them difficult to directly apply to multicore CPU evaluations. This section provides a summary of key challenges for performance measurement and then discusses popular metrics that overcome these challenges. It also briefly discusses metrics related to power and energy, given their emphasis throughout the lifetime of multicore CPUs.

Multithreaded Application Performance During the evaluation of multithreaded applications, an important challenge is factoring in the synchronization overhead. Due to the nondeterministic nature of synchronization, multiple runs of the same program on the same machine, with identical data sets and an identical number of threads, can have different total runtimes and can have different total instruction counts (e.g., a thread spinning as it waits to acquire a lock will execute additional instructions for each time it checks the lock variable). As a result, it is important to (1) compare equivalent amounts of work performed, as opposed to specific instruction counts (especially in architectural simulators that tend to execute only parts of a program) and (2) execute the configuration for each data point multiple times and report a mean (ideally with error bars) instead of the results of a single run.

For a fixed amount of work (e.g., an entire application, a multithreaded kernel such as an entire parallel section of an application), the best measurement for comparison is the total wall clock execution time (i.e., the time from when the application/kernel starts, to the time the *last* thread finishes). While CPU cycle counts can act as a proxy for execution time, one must take care of using the global cycle count and not a per-thread cycle count that might not track time during which a thread went to sleep (e.g., to wait for a lock). To understand the benefits of parallelism when a program uses N CPU cores, a common metric that is used is *parallel speedup*, $S(N)$:

$$S(N) = \frac{T_s}{T_p} \tag{4}$$

where T_p is the total execution time of the parallel version of the program for N cores, and T_s is the total execution time of the *sequential* version of the program (and not the parallel version with one core). The equation uses the sequential version of the program to ensure that parallel speedup captures the overheads of synchronization. As a result, $S(1)$ can often be less than 1. Note that unlike with single-thread applications, the IPC (*instructions per cycle*) of a program should not be used as a substitute for execution time, as IPC values can be skewed by synchronization traffic and other nondeterministic behavior.

A related metric is *parallel efficiency*, $E(N)$:

$$E(N) = \frac{T_s}{T_p \times N} \tag{5}$$

For parallel efficiency, a value of 1 indicates no overheads due to parallelization (i.e., the application is making full use of all cores), whereas values significantly lower than 1 indicate high overheads due to issues such as synchronization or serial execution.

Multiprogrammed Workload Performance *Multiprogrammed workloads* (i.e., a bundle of independent, concurrently executing applications) are unable to employ the performance metrics used by multithreaded applications because the different applications are working to complete separate tasks. As a result, the applications in a workload can exhibit significant heterogeneity. For example, a four-application workload may have two of the applications that are compute-intensive (and thus have high IPCs), while the other two applications are memory intensive (and thus have low IPCs). If one were to use an aggregate metric of total execution time or the sum of IPCs (e.g., for comparing several potential system improvements to decide which one to implement), the metric may unfairly bias improvements to one class of applications (e.g., compute-intensive applications) over the other. To avoid this bias when measuring overall system performance for a multiprogrammed workload, it is important to use metrics that attempt to *normalize* the relative benefits to each application though there are several ways to perform this normalization (Eyerman and Eechkout 2008).

Related to this is the issue of *fairness*. As the applications in a multiprogrammed workload are independent of each other, one application's resource usage can generate interference that slows down the other applications that are sharing the multicore CPU, compared to if those applications ran alone on the CPU. For any one application in the workload, its *slowdown* is defined in terms of its performance when running *alone* on the CPU compared to its performance when the CPU is *shared* with other applications (Kim et al. 2004; Mutlu and Moscibroda 2006):

$$slowdown = \frac{IPC_{alone}}{IPC_{shared}} \tag{6}$$

(To simplify the discussion of multiprogrammed workload metrics, this section assumes that each application is single-threaded, which allows us to use IPC in the equations (as there is no synchronization overhead). For multithreaded applications in a multiprogrammed workload, IPC should be replaced by total execution time for that application.) By using a ratio of IPCs, the slowdown metric normalizes away the inherent compute-intensive or memory-intensive behavior of each application, and reports what fraction of the application's performance was lost due to interference. A system is *unfair* if it slows down some applications significantly more than others.

There are two metrics that can quantify (un)fairness. The first is *maximum slowdown*, which simply says that the worst slowdown experienced by any one application is an indication of unfairness due to interference and that a smaller maximum slowdown is more equitable (Das et al. 2009). The second defines fairness as the ratio between the best slowdown and worst slowdown experienced by applications in the workload (Gabor et al. 2006; Eyerman and Eechkout 2008):

$$fairness = \min_{i,j} \frac{slowdown_i}{slowdown_j} = \frac{\min_i slowdown_i}{\max_j slowdown_j} \tag{7}$$

where i and j are the members of the set of all applications in the workload. Informally, a fairness of 1 indicates that all applications are experiencing an equal slowdown, while at the other extreme, a fairness of 0 indicates that at least one application is experiencing starvation. Note that unfairness in this case is defined as the inverse of fairness.

Aggregate performance metrics for multiprogrammed workloads incorporate some notion of both overall system throughput and fairness, in an attempt to remove the bias mentioned above that can arise from using absolute IPCs. A popular metric is weighted speedup (WS) (Snavely and Tullsen 2000; Eyerman and Eechkout 2008), which sums up the normalized speedups (i.e., the inverse of slowdown) of each application i in the workload to represent system throughput:

$$WS = \sum_i \frac{IPC_{shared}}{IPC_{alone}} = \sum_i \frac{1}{slowdown_i} \tag{8}$$

A larger value of WS is better, as it indicates higher system throughput (i.e., lower aggregate impacts of interference). Note that for an n-application workload, WS typically ranges between 0 and n, indicating that the metric is dependent on the number of applications (and often, by proxy, the number of cores), as this represents a throughput for a specific system. As a result, when comparing two systems and reporting *improvements* due to a system modification, one should typically report a ratio of weighted speedups (occasionally referred to as *WS improvement*, although this is sometimes confusingly reported as just WS in several papers):

$$WS \text{ improvement} = \frac{WS_{after_modification}}{WS_{before_modification}} \tag{9}$$

There is some debate about whether *WS* effectively captures unfairness, which has led several researchers to use the *harmonic mean* (*HM*) of speedups (Luo et al. 2001; Eyerman and Eechkout 2008) (sometimes referred to as *harmonic speedup*) in place of or in addition to *WS*:

$$HM = \frac{n}{\sum_i \frac{IPC_{alone}}{IPC_{shared}}} = \frac{n}{\sum_i slowdown_i} \tag{10}$$

HM represents the average slowdown in a user response (i.e., the turnaround time for an output produced by the application) for each application in the workload due to interference (Eyerman and Eechkout 2008). Like with WS, higher is better, and to compare two systems and report an improvement, one should calculate the ratio of HM values for the two systems.

Power and Energy While power and energy are related, they represent different limiting factors experienced by modern computers. Power (P) represents a *rate* of work being completed and can be calculated as a function of current (I) and voltage (V):

$$P = I \times V \tag{11}$$

At a high level, power in a CPU can be broken down into a *dynamic* component (e.g., the power consumed due to the active switching of transistors to perform work, short-circuit power consumed during gate switching when transistors temporarily connect the high voltage rail to ground due to transistor timing variation) and a *static* component (e.g., the leakage of power due to imperfections in the switching behavior of a transistor). The section titled "Optimizing CPU Cores for Parallelism" briefly discusses components that impact dynamic power. Note that in the past, dynamic power was orders of magnitude larger than static power, so static power was thought of as a trivial factor in total power consumption. Today, because decades of Dennard Scaling translated to significant dynamic power reductions, static power makes up a nontrivial fraction of total CPU power. Power consumption has a direct correlation with thermal dissipation and is used as a proxy to quantify the heat generated by the CPU. The *areal power density*, which divides the power by the surface area of the CPU die, is used to determine how aggressive thermal cooling solutions (e.g., heatsinks, liquid cooling, fans) need to be to remove dissipated heat from the CPU and keep the die within safe thermal operating limits. While areal power density is used at design time to provision heat dissipation capacity, the CPU makes use of *temperature* readings from multiple sensors embedded at various locations in both the CPU chip and the motherboard to dynamically control heat dissipation management, including cooling intensity (e.g., fan speed) and CPU power throttling.

While power consumption was the key concern during the early years of multicore CPUs, energy emerged as a first-order concern during the 2010s. Energy

(E) is the total electrical cost of performing a given amount of work and can be calculated as a function of power:

$$E = P \times t \tag{12}$$

where t is the time required to complete the defined amount of work. Challenges associated with two extreme ends of computing platforms have resulted in the growing emphasis on energy consumption, in addition to power consumption. First, portable computers such as laptops and smartphones are battery-constrained, as their available uptime depends on the total energy capacity of the computer's battery and the amount of energy that the system (including the CPU) consumes for running applications. Second, the large number of servers in data centers and cloud computing environments can result in exorbitant financial and environmental costs to provide enough energy to perform user services. In both cases, reducing the total energy used for a given application can result in more availability at a lower overall cost. A related metric of interest is *energy efficiency*, which summarizes the energy used for a single operation (e.g., an instruction, a microkernel), though one downside of energy efficiency is the difficulty of defining an operation in an equal way across platforms (e.g., two CPUs with different ISAs may not have equivalent instructions). Several modern multicore CPUs contain ISA-compatible cores of heterogeneous size and capability (see "Heterogeneous CPU Cores"), and dynamic energy and/or energy efficiency metrics based on the characteristics of a thread are used to select which of the heterogeneous cores will execute the thread.

The Evolution of Multicore CPUs

In the years that have elapsed since the introduction of multicore CPUs, there have been a number of innovations to the general microarchitecture described in "Multicore CPU Hardware Design." While some of these innovations have been limited to specific manufacturers or models, others have become commonplace across modern CPUs. This section highlights three of the most significant shifts in multicore CPU design and leaves the exploration of other innovations as an exercise for the reader. These three shifts are finding widespread acceptance in contemporary multicore CPUs: (1) the integration of specialized components on-chip alongside the general-purpose CPU cores into what are known as *systems-on-chip* (SoCs), (2) the diversification of the constituent cores in a multicore CPU, and (3) the advent of composable *chiplets* that can allow for the easy integration of many smaller silicon dies in a single chip.

Systems-on-Chip

Just as the limits of areal power density and thermal dissipation were a key motivator for the rise of multicore CPUs, a new pressure point that came to

prominence a few years later ushered in the next key change. The emergence of the smartphone drove a need to reduce total energy consumption, given the limited battery capacities that were available in a portable form factor. To maximize energy efficiency, smartphones made use of *systems-on-chip* (SoCs). A system-on-chip tightly integrates many different components, which conventionally would have been implemented using multiple chips for desktop and server computers, into a single chip. Early SoC examples date back to the mid-1970s, such as the Intel 5810 (Intel Corp. 1976) introduced in 1974, and were designed to minimize battery consumption in then-new electronic wristwatches. Over time, platform-specific SoCs became relatively commonplace in the embedded systems community. Early smartphones such as the original Apple iPhone, from 2007, made use of Samsung SoCs that contained a single Arm CPU core, a graphics processing unit (GPU), and caches integrated onto a single chip (Mannion 2007).

As the functionality and ubiquity of the smartphone expanded, their underlying SoCs incorporated significantly more components, including more CPU cores. At a high level, the goal of these additional components is to introduce *specialization* for commonly performed operations, in order to significantly improve the efficiency of these operations compared to executing them on a general-purpose CPU core. Figure 14 shows several key components of the Apple A17 Pro SoC (Apple Inc. 2023), which started production in 2023 for use in the iPhone 15 Pro series of smartphones. As the figure shows, the SoC contains six CPU cores (which are heterogeneous; see "Heterogeneous CPU Cores"), a GPU, multiple fixed-function accelerators (a neural engine for machine learning inference, an image signal processor for photo processing, a video codec engine for video recording and streaming, a display engine for screen image generation), I/O interfaces (including a dedicated USB controller), a system-level cache (an LLC that is available to the CPU, GPU, and all accelerators), and four LPDDR5X memory controllers.

The process of identifying which components to include in the SoC (beyond the basic CPU and GPU) makes use of *profiling tools*. These tools can monitor the performance (and often the energy) of an existing chip as one or more applications execute. To do this, modern profiling tools make use of hardware *performance counters*, which are registers built into the chip logic to track various events taking place during execution. For SoC design, profiling can identify the applications or application kernels that are bottlenecked by the existing chip, which become candidates for acceleration using dedicated SoC components. Of these candidates, a subset of them is chosen for dedicated acceleration based on a combination of factors, including the frequency of application/kernel usage, the area required for a fixed-function accelerator, available chip area, power and energy budgets, and the availability of existing accelerator designs (including the availability of third-party designs known as *IP cores*, where IP stands for intellectual property). Note that even among state-of-the-art SoCs that target the same platform, the exact components can vary. As one example, Qualcomm's Snapdragon 8 Gen 3 SoC (Qualcomm Technologies 2024) for smartphones integrates a 5G cell modem, Wi-Fi and Bluetooth transceivers, and security accelerators in the chip and makes use of a different combination of heterogeneous cores than the A17 Pro.

Heterogeneous CPU Cores

As discussed in "Systems-on-Chip," a key motivator for introducing fixed-function accelerators in SoCs is to improve energy efficiency for commonly performed operations. Even with these accelerators, there remains a need to execute code using a general-purpose CPU core (e.g., less common operations, OS system calls, and irregular workloads that are more difficult to accelerate). Using the smartphone as a motivating example, the CPU cores in a smartphone SoC have seen significantly increasing complexity in their designs (e.g., more sophisticated predictors, deeper speculation, and larger superscalar widths (Grayson et al. 2020)), to meet the increasing demands of smartphone applications that have become more sophisticated over the years. While larger CPU cores can enable efficient execution of modern compute-heavy applications compared to smaller cores, the increased core complexity (along with associated increases in leakage and clock power) becomes more *inefficient* for more memory-bound applications that cannot make effective use of these additional resources.

A seminal study in 2003 illustrated how having a *heterogeneous* combination of large cores *and* small cores in a single multicore CPU is significantly more energy efficient than the use of homogeneous (i.e., identical) cores (Kumar et al. 2003). If all of the cores maintain the same instruction set architecture (ISA), an application can migrate from a large core to a smaller core when it enters a more memory-bound phase and can migrate back to the larger core when it enters a more compute-bound phase. Arm introduced their first CPU cores designed for a heterogeneous multicore CPU in 2011, under the name big.LITTLE (Greenhalgh 2011). The first big.LITTLE pairing combined large Cortex-A15 cores with small Cortex-A7 cores, where the Cortex-A15 cores could achieve up to 3.0× the performance of the Cortex-A7 cores, but the Cortex-A7 could achieve up to 3.8× the energy efficiency, for selected benchmarks. For what Arm defined as low- to mid-range workloads, the Cortex-A7 was expected to execute these workloads with a dramatic reduction in power consumption, with no expected loss in performance compared to the Cortex-A15. Initially, core heterogeneity was hidden from the system software, with the CPU hardware transparently migrating applications between a big core and a paired little core. This meant that for any big–little pair, one of the cores was idle. Today, core heterogeneity is exposed to the OS scheduler, as discussed in "Optimizing Operating Systems for Multicore CPUs."

Since the commercial introduction of heterogeneous CPU cores, they have become commonplace in modern multicore CPUs. The example A17 Pro CPU from Apple, shown in Fig. 14, contains two types of CPU cores (Apple Inc. 2023): (1) a pair of large high-performance cores and (2) four smaller high-efficiency cores. Qualcomm's Snapdragon 8 Gen 3 CPU contains three distinct types of CPU cores (Qualcomm Technologies 2024): (1) a single large prime core that runs at a frequency of up to 3.4 GHz, (2) five performance cores, large but notably smaller than the prime core, that run at a frequency of up to 3.2 GHz, and (3) two small efficiency cores that run at a frequency of up to 2.3 GHz. While the usage of

heterogeneous CPU cores first became popular for mobile SoCs, it can now be seen in a wide range of modern multicore CPUs. For example, Intel's 12th generation of Core CPUs introduced heterogeneous CPU cores (named P-cores and E-cores) for desktop computers (Rotem et al. 2021).

Chiplet-Based Multicore Design

Beyond the cores themselves, the success of SoCs have demonstrated the benefits of maximizing energy efficiency through directed specialization. However, there remains a tension between high degrees of specialization and *non-recurring engineering* (NRE) costs, such as those involved with design, layout, and verification. As a simple motivating example, let us revisit the tiled multicore design from "Optimizing CPU Cores for Parallelism." While tiling helps reduce NRE costs, the tile is a fixed design: For one core, there is a fixed amount of L1 instruction and data caches, L2 cache, and potentially LLC slice. If a manufacturer wants to adapt this tile for a platform whose workloads do not exhibit significant locality, they may want to significantly reduce the cache sizes, but doing so requires a new tile to be designed and verified. Moreover, the die that is etched for a multicore CPU will

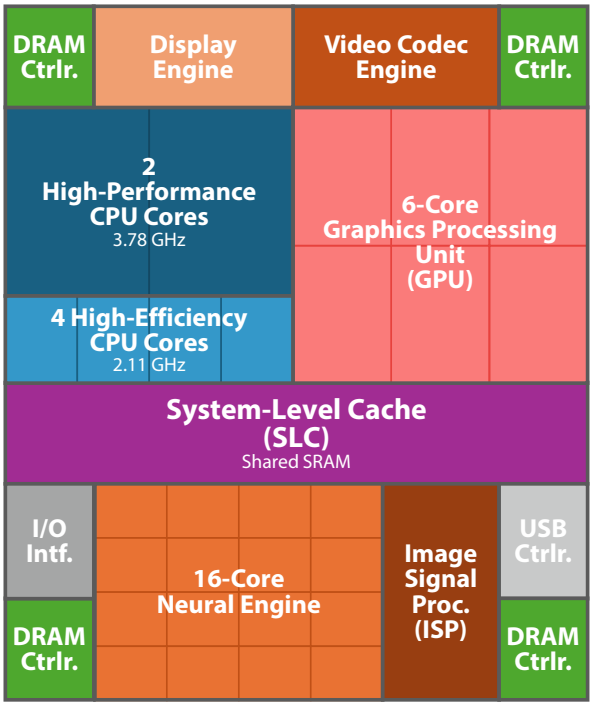


Fig. 14 Selected components in the Apple A17 Pro system-on-chip. Note that components may not be to scale with each other

have a fixed number of tiles laid out, again restricting the flexibility of the CPU and requiring nontrivial NRE costs if a die with a different core/tile count is needed.

The advent of *chiplets* provides a new way to compose a multicore CPU in a more modular fashion, avoiding some of these NRE costs. A chiplet is a small die that contains a subset of the functionality that would be contained in a standard die. Instead of laying out a multicore CPU design using a monolithic die, designers can design smaller chiplets with individual components, such as cores or caches. For example, in place of a single die containing eight cores and their associated caches, a chip for a multicore CPU could be composed using eight core chiplets, eight L1 cache chiplets, eight L2 cache chiplets, and 16 LLC slice chiplets. If the manufacturer now wants a multicore CPU with fewer cores and larger caches, they can reuse the chiplets to compose a chip with two core chiplets, 16 of the L1 and L2 cache chiplets each, and 64 LLC slice chiplets. While chiplets are one example of a broader concept called *multi-chip modules* (MCMs), which has been around for decades, it was conventionally difficult to have more than a handful of dies in an MCM due to packaging costs and alignment issues. Recent advances in *interposer* design, where an interposer provides a substrate with many short-distance wires to connect dies together, have reduced manufacturing costs, complexity, and faults for assembling many dies in one MCM.

Chiplets offer four advantages for manufacturing. First, as already discussed, chiplets allow for modular components that can be reused and resized after the dies have been fabricated, at low cost. Second, chiplets can allow for dies fabricated using different manufacturing process technologies to be connected together into a single package (this is known as *heterogeneous integration*). If, for example, the cache does not need to be manufactured using the state-of-the-art manufacturing process, manufacturers can reduce costs by fabbing the chip using an older, cheaper process. Third, overall yield increases, because a silicon fault is now isolated to a much smaller chiplet, which can be replaced at much lower cost than disposing of an entire die. Fourth, with the breakdown of Dennard scaling (Dennard et al. 1972, 1974), dies are now growing in size to continue scaling up the total transistor count, but these sizes are approaching the reticle limits (i.e., the largest possible chip that can be etched) of our lithography equipment. Chiplets can overcome these limits by allowing for multiple larger chiplet dies to be composed into a package, where the total area of the chiplets is significantly larger than what any one die could be.

Several manufacturers have started incorporating chiplet-based design for multicore CPUs. AMD has been responsible for significant innovation in the area of chiplets and interposer design and has been manufacturing chiplet-based multicore CPUs starting with the first-generation EPYC CPUs in 2017 (Naffziger et al. 2021). Figure 15 shows die shots of the AMD EPYC 7702 CPU, released in 2019, which consists of nine chiplets: eight core complex dies (CCDs), fabricated in a 7 nm process, with eight cores (and their private caches and LLC slices) per CCD, and a single I/O die in the center, fabricated in a 14 nm process, with memory and I/O controllers. Apple and Intel have also announced the incorporation of chiplet-based design into their latest multicore CPUs (Smith et al. 2022; Rodgers et al. 2024).

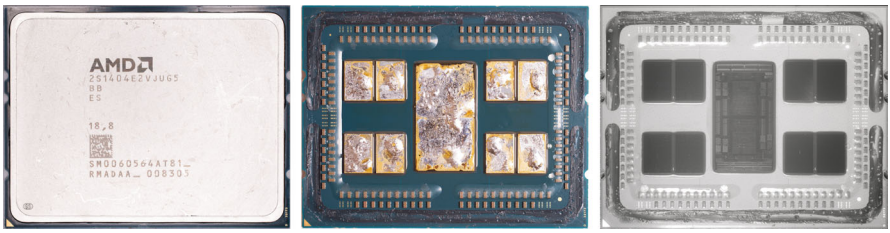


Fig. 15 Lidded, delidded, and infrared views of the AMD EPYC 7702 CPU and its chiplets (Fritz 2019)

Conclusion

The rise of multicore CPUs highlighted a key shift in the computer architecture community, as concerns about thermal dissipation and limitations of ILP hastened a collective change in mindset about the importance of power and the potential of parallel processing. Innovations in multicore CPU design have led to reduced design and verification effort, support for specialized hardware accelerators, and increased composability of modular components. Today, multicore CPUs can be found across most modern computers, ranging from embedded platforms and smartphones, through personal laptops and desktops, to large-scale distributed computing environments. Combined with the emergence of simplified parallel programming frameworks, multicore CPUs have led to commonplace exploitation of thread-level parallelism, delivering significant performance improvements while maintaining reasonable power and energy budgets. Multicore CPUs are expected to continue evolving over the next several decades, as recent trends toward CPU specialization (particularly in the modern CPU landscape with systems-on-chip and chiplet-based fabrication) open up new opportunities for maximizing the efficiency and performance of the next generation of computing platforms.

Acknowledgments The author thanks Ryan Wong, Sudhanshu Agarwal, Yiqiu Sun, and Minh S. Q. Truong for reviewing multiple versions of this chapter and providing helpful feedback.

References

Amdahl GM (1967) Validity of the single processor approach to achieving large-scale computing capabilities. In: SJCC

Anderson JP, Hoffman SA, Shifman J, Williams RJ (1962) D825 – a multiple-computer system for command & control. In: FJCC

Apple Inc. (2023) Apple event, 12 Sept 2023. https://www.apple.com/apple-events/

Backes L, Jiménez DA (2019) The impact of cache inclusion policies on cache management techniques. In: MEMSYS

Balasubramonian R (2019) Innovations in the memory system. Springer Nature Switzerland

Bitirgen R, İpek E, Martínez JF (2008) Coordinated management of multiple interacting resources in chip multiprocessors: a machine learning approach. In: ISCA
Boggs D, Baktha A, Hawkins J, Marr DT, Miller JA, Roussel P, Singhal R, Toll B, Venkatraman K (2004) The microarchitecture of the Intel® Pentium® 4 Processor on 90 nm Technology. Intel Technol J
Censier LM, Feautrier P (1978) A new solution to coherence problems in multicache systems. IEEE Trans Comput
Corbató FJ, Merwin-Daggett M, Daley RC (1962) An experimental time-sharing system. In: SJCC
Das R, Mutlu O, Moscibroda T, and Das CR (2009) Application-aware prioritization mechanisms for on-chip networks. In: MICRO
De V, Borkar S (1999) Technology and design challenges for low power and high performance. In: DAC
Dekker TJ (2022) History of Dekker's algorithm for mutual exclusion. In: Alberts G, Groote JF (eds) Tales of electrologica. Springer, Cham
Dennard RH, Gaensslen FH, Kuhn L, Yu H-N (1972) Design of micron MOS switching devices. In: IEDM
Dennard RH, Gaensslen FH, Yu H-N, Rideout VL, Bassous E, LeBlanc AR (1974) Design of ion-implanted MOSFET's with very small physical dimensions. J Solid-State Circuits
Dijkstra EW (1962) Over de sequentialiteit van procesbeschrijvingen. EWD35, circulated privately
Dijkstra EW (1968) Cooperating sequential processes. In: Genuys F (ed) Programming languages: NATO Advanced Study Institute. Academic Press
Dreyfus P (1958) System design of the Bull Gamma 60. In: WJCC
Dubois M, Scheurich C, and Briggs FA (1986) Memory access buffering in multiprocessors. In: ISCA
Dunwell SW (1956) Design objectives for the IBM Stretch computer. In: EJCC
Esmaeilzadeh H, Blem E, Amant RS, Sankaralingam K, Burger D (2011) Dark silicon and the end of multicore scaling. In: ISCA
Eyerman S, Eechkout L (2008) System-level performance metrics for multiprogram workloads. IEEE Micro
Farooqi MN, Abduljabbar M, Beltran V, Teruel X, Ferrer R, Martorell X, Pericàs M (2022) Parallel programming models. In: Chattopadhyay A (ed) Handbook of computer architecture. Springer Nature Singapore
Fillo M, Keckler SW, Dally WJ, Carter NP, Chang A, Gurevich Y, Lee WS (1995) The M-Machine multicomputer. In: MICRO
Flynn MJ (1966) Very high-speed computing systems. Proc IEEE
Frank DJ, Dennard RH, Nowak E, Solomon PM, Taur Y, Wong H-SP (2001) Device scaling limits of Si MOSFETs and their application dependencies. Proc IEEE
Fritz F (2019) AMD/Zen2/EPYC/Rome. https://www.flickr.com/photos/130561288@N04/albums/72157715067973156. Photo album
Gabor R, Weiss S, Mendelson A (2006) Fairness and throughput in switch on event multithreading. In: MICRO
Goodman JR (1983) Using cache memory to reduce processor-memory traffic. In: ISCA
Grayson B, Rupley J, Zuraski Jr G, Quinnell E, Jiménez DA, Nakra T, Kitchin P, Hensley R, Brekelbaum E, Sinha V, Ghiya A (2020) Evolution of the Samsung Exynos CPU microarchitecture. In: ISCA
Greenhalgh P (2011) Big.LITTLE processing with ARM Cortex™-A15 & Cortex-A7. ARM Ltd., White Paper
Gustafson JL (1988) Reevaluating Amdahl's law. Commun ACM
Hanawa M, Nishimukai T, Nishii O, Suzuki M, Yano K, Hiraki M, Shukuri S, Nishida T (1991) On-chip multiple superscalar processors with secondary cache memories. In: ICCD
Herdrich A, Verplanke E, Autee P, Illikkal R, Gianos C, Singhal R, Iyer R (2016) Cache QoS: from concept to reality in the Intel® Xeon® Processor E5-2600 v3 product family. In: HPCA
Hinton G (2010) Key Nehalem Choices, presentation at Stanford University

Huang M, Mehalel M, Arvapalli R, He S (2013) An energy efficient 32-nm 20-MB shared on-die L3 cache for Intel® Xeon® Processor E5 family. In: JSSC
Intel Corp. (1976) 5810A Single Chip LCD Time/Seconds/Date Watch Circuit, datasheet. In: Data catalog
Intel Corp. (1989) i486 Microprocessor, datasheet, Order Number 240440-002
Intel Corp. (1999) Intel Pentium III Processor 600 MHz, 512K Cache, 100 MHz FSB. https://ark.intel.com/content/www/us/en/ark/products/27545/intel-pentium-iii-processor-600-mhz-512k-cache-100-mhz-fsb.html
Intel Corp. (2004) Intel Pentium 4 Processor 570J Supporting HT Technology. https://ark.intel.com/content/www/us/en/ark/products/27475/intel-pentium-4-processor-570j-supporting-ht-technology-1m-cache-3-80-ghz-800-mhz-fsb.html
Intel Corp. (2006) Intel Core 2 Duo Processor E6700. https://ark.intel.com/content/www/us/en/ark/products/27251/intel-core-2-duo-processor-e6700-4m-cache-2-66-ghz-1066-mhz-fsb.html
JEDEC Solid State Technology Assn. (2020) JESD235C: High Bandwidth Memory (HBM) DRAM
JEDEC Solid State Technology Assn. (2024) JESD79-5C: DDR5 SDRAM Standard
Joyce TF, Kelly RP, Shen J-K, Raguin MM (1987) Multiprocessors on a Single Semiconductor Chip. U.S. Patent 4 942 547
Kim C, Burger D, Keckler SW (2002) An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In: ASPLOS
Kim S, Chandra D, Solihin Y (2004) Fair cache sharing and partitioning in a chip multiprocessor architecture. In: PACT
Kim W, Gupta MS, Wei G-Y, Brooks D (2008) System level analysis of fast, per-core DVFS using on-chip switching regulators. In: HPCA
Kroft D (1981) Lockup-free instruction fetch/prefetch cache organization. In: ISCA
Kumar R, Farkas KI, Jouppi NP, Ranganathan P, Tullsen DM (2003) Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction. In: MICRO
Lamport L (1979) How to make a multiprocessor computer that correctly executes multiprocess program. IEEE Trans Comput
Leiner AL (1952) System Specifications for the DYSEAC. U.S. Nat'l. Bureau of Standards, Tech. Rep.
Lempel O (2011) 2nd Generation Intel® Core™ Processor Family: Intel® Core™ i7, i5 and i3. In: Hot Chips
Linux Kernel Organization, Inc. (2023a) Capacity Aware Scheduling. In: The Linux Kernel Documentation, https://docs.kernel.org/scheduler/sched-capacity.html
Linux Kernel Organization, Inc. (2023b) Energy Aware Scheduling. In: The Linux Kernel Documentation. https://docs.kernel.org/scheduler/sched-energy.html
Longergan W, King P (1961) Design of the B 5000 System. Datamation, May 1961
Luo K, Gummaraju J, Franklin M (2001) Balancing throughput and fairness in SMT processors. In: ISPASS
Macken P, Degrauwe M, Van Paemel M, Oguey H (1990) A voltage reduction technique for digital systems. In: ISSCC
Mannion P (2007) Under the Hood: Inside the Apple iPhone. EE Times
Mattson TG, Sanders BA, Massingill BL (2004) Patterns for parallel programming. Addison-Wesley Professional
Menabrea LF (1842) Notions sur la machine analytique de M. Charles Babbage. Bibliothèque universelle de Genève
Minneapolis–Honeywell DATAmatic Division (1960) Honeywell 800 Programmers' Reference Manual
Moore GE (1965) Cramming more components onto integrated circuits. Electronics
Moore GE (1975) Progress in digital integrated electronics. In: IEDM
Mutlu O, Moscibroda T (2006) Stall-time fair memory access scheduling for chip multiprocessors. In: MICRO

Naffziger S, Beck N, Burd T, Lepak K, Loh GH, Subramony M, White S (2021) Pioneering chiplet technology and design for the AMD EPYC™ and Ryzen™ processor familie. In: ISCA
Nagarajan V, Sorin DJ, Hill MD, Wood DA (2020) A primer on memory consistency and cache coherence, 2nd edn. Springer Cham
National Semiconductor Corp. (1982) COP2440/COP2441/COP2442 and COP2340/COP2341/COP2342 Single-Chip Dual CPU Microcontrollers, datasheet. In: COPS microcontrollers databook
Olukotun K, Nayfeh BA, Hammond L, Wilson K, Chang K (1996) The case for a single-chip multiprocessor. In: ASPLOS
Papamarcos MS, Patel JH (1984) A low-overhead coherence solution for multiprocessors with private cache memories. In: ISCA
Qualcomm Technologies, Inc. (2024) Snapdragon 8 Gen 3 Mobile Platform, product brief
Rixner S, Dally WJ, Kapasi UJ, Mattson P, Owens JD (2000) Memory access scheduling. In: ISCA
Rodgers L, Clark D, Joiner S, Haslett B, de la Torre Arenas I, Learner S (2024) Inside the miracle of modern chip manufacturing. Financ Times
Rojas R (1996) Konrad Zuse's legacy: the architecture of the Z1 and Z3. IEEE Ann Hist Comput
Ronen R, Mendelson A, Lai KK, Lu S-L, Pollack FJ, Shen JP (2001) Coming challenges in microarchitecture and architecture. Proc IEEE
Rotem E, Mandelblat Y, Basin V, Weissmann E, Gihon A, Chabukswar R, Fenger R, Gupta M (2021) Alder Lake architecture. In: Hot chips
Schmidt U, Caesar K (1991) Datawave: a single-chip multiprocessor for video applications. IEEE Micro
Smith AJ (1982) Cache memories. ACM Comput Surv
Smith MS (2022) Single-chip processors have reached their limits. IEEE Spectr (2022)
Snavely A, Tullsen DM (2000) Symbiotic jobscheduling for a simultaneous multithreaded processor. In: ASPLOS
Sohi GS, Breach SE, Vijaykumar T (1995) Multiscalar processors. In: ISCA
Sohi GS, Franklin M (1991) High-bandwidth data memory systems for superscalar processors. In: ASPLOS
SPARC International Inc. (1991) The SPARC architecture manual, version 8
Taub AH, Gillies DB, Meagher RE, Muller DE, McKay RW, Nash JP, Poppelbaum WJ, Robertson JE (1957) On the design of a very high-speed computer. University of Illinois Digital Computer Laboratory, Tech. Rep. 80
Tendler JM, Dodson JS, Fields Jr JS, Le H, Sinharoy B (200) POWER4 system microarchitecture. IBM J Res Dev
Thornton JE (1964) Parallel operation in the Control Data 6600. In: FJC
Torrellas J, Lam MS, Hennessy JL (1990) Shared data placement optimizations to reduce multiprocessor cache miss rates. In: ICPP
Waingold E, Taylor M, Srikrishna D, Sarkar V, Lee W, Lee V, Kim J, Frank M, Finch P, Barua R, Babb J, Amarasinghe S, Agarwal A (1997) Baring it all to software: Raw machines. Computer
Whitney DC, White Jr CH (1968) Time-sharing services. Mod Data Syst
Witt BI (1966) The functional structure of OS/360, part II: Job and task management. IBM Syst J
Wysocki RJ (2020) CPU performance scaling. In: The Linux kernel documentation. https://docs.kernel.org/admin-guide/pm/cpufreq.html.
Zuse K (1949) Rechenmaschine zur Durchfuehrung von arithmetischen Rechenoperationen. German Patent 975 966, 30 Jun 1949

Part IV
Emerging Computing Architectures

Compute-in-Memory Architecture

19

Hongwu Jiang, Shanshi Huang, and Shimeng Yu

Contents

Introduction 648
DNN Basics and Corresponding CIM Principle 650
Architecture and Algorithm Techniques for CIM 652
  Hierarchical Architecture of CIM 652
  Network Mapping Strategies 653
  Pipeline Design in CIM Architecture 660
  Quantization Techniques in CIM Architectures 664
Hardware Implementations for CIM Architecture 667
  Device Technologies 667
  Overcoming the Non-idealities from eNVM 671
  Circuit Techniques for CIM 672
  Output Sensing 675
Frameworks for Evaluating CIM Designs 681
Conclusion 682
References 683

Abstract

In the era of big data and artificial intelligence, hardware advancement in throughput and energy efficiency is essential for both cloud and edge computations. Because of the merged data storage and computing units, compute-in-memory is becoming one of the desirable choices for data-centric applications to mitigate the memory wall bottleneck in von-Neumann architecture. In this chapter, the recent architectural designs and underlying circuit/device technologies for compute-in-memory are surveyed. The related design challenges and

H. Jiang · S. Huang · S. Yu (✉)
School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, USA
e-mail: hjiang318@gatech.edu; shuang406@gatech.edu; shimeng.yu@ece.gatech.edu

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_62

prospects are also discussed to provide an in-depth understanding of interactions between algorithms/architectures and circuits/devices. The chapter is organized hierarchically: the overview of the field (Introduction section); the principle of compute-in-memory (section "DNN Basics and Corresponding CIM Principle"); the latest architecture and algorithm techniques including network model, data flow, pipeline design, and quantization approaches (section "Architecture and Algorithm Techniques for CIM"); the related hardware support including embedded memory technologies such as static random access memories and emerging nonvolatile memories, as well as the peripheral circuit designs with a focus on the analog-to-digital converters (section "Hardware Implementations for CIM Architecture"); a summary and outlook of the compute-in-memory architecture (Conclusion section).

Keywords

Compute-in-memory · Deep neural network · Hardware accelerator · SRAM · eNVMs · ADC

Introduction

Hardware development is a key facilitator for the rapid growth of data-centric applications, in particular for artificial intelligence (AI)-related tasks. Those heavy-load computations are traditionally performed by central processing units (CPUs) or graphic processing units (GPUs). Recently, application-specific processors and accelerators have also been developed based on silicon complementary metal-oxide-semiconductor (CMOS) technology (e.g., tensor processing unit (TPU) (Jouppi et al. 2017), Eyeriss (Chen et al. 2016)). However, in the big data era, data movements and speed discrepancy between computing and storage units become the bottleneck of conventional von Neumann architectures, known as the memory wall. To this end, compute-in-memory (CIM), where information can be processed and stored at the same locations, is emerging as an efficient paradigm to address the memory wall bottleneck (Yu 2018).

CIM (or processing-in-memory, PIM) is not a new concept, which was first proposed back in the 1970s, and there have been several subsequent works exploring different ways to integrate computation and memory on standalone dynamic random access memory (DRAM) since the 1990s (e.g., Computational RAM (Elliott et al. 1999)). Unfortunately, the idea of integrating logic with memory banks was impractical due to the incompatibility of the DRAM and logic fabrication processes at that time. Driven by the recent 3D heterogeneous integration technology, PIM architectures are recently resurgent on standalone memory systems (Ueyoshi et al. 2018). However, to realize the seamless integration between logic and memory components, the through-silicon-via (TSV) pitch needs to approach the back-end-of-line (BEOL) pitch in the sub-μm range, requiring further technology advances. Therefore, this chapter is focused on embedded on-chip memories for CIM implementations.

In general, CIM-based approaches have two advantages: (1) improve the computational efficiency of a range of different functions, such as Boolean logic operations (e.g., AND, OR, XNOR), simple arithmetic operations (e.g., addition, multiplication), and linear algebra operations (e.g., dot products, matrix multiplication), following the benefits from FPGA-based structures (DeHon 2000); (2) save the time and energy by reducing the amount of data transfer. The benefits of reduced data transfer are modeled and estimated in Ronen et al. (2022). The demonstrations of CIM have been applied to various applications, ranging from scientific computing, digital image processing, security, and spiking neural network (SNN) to deep learning inference/training. For instance, scientific computing such as solving linear and partial differential equations could be implemented by CIM to reduce the computational complexity while usually requiring a high-precision scheme and low variability to ensure computational accuracy (Feng et al. 2021). Besides, many digital image processing techniques such as signal filtering and image transformation also include a large number of VMM operations, which can be accelerated by in-memory computing (Li et al. 2018a). Moreover, CIM-based image processing also shows potential in edge preliminary processing in analog for fast speed and low energy consumption (Zhu et al. 2021). CIM with device variations can be exploited to design strong physically unclonable function for security purpose (Gao et al. 2016). Another application domain for CIM is neuromorphic computing. For example, the conductance tuning of memory devices could be used to imitate synapse behavior such as spike-timing-depend plasticity (STDP) (Kim et al. 2020). Lastly, the most representative and important application of CIM is deep learning (DL) acceleration. CIM implementation for DL could benefit from both the efficiency of parallel computing and reduced memory access. Hence, in this chapter, DL is chosen as an illustration application to provide a broad overview of CIM. In the recent decade, DL algorithms, such as convolutional neural network (CNN), have achieved remarkable success in various AI applications. The state-of-the-art DL algorithms require a large memory capacity as the size of deep neural networks (DNNs) increases dramatically (e.g., ViT-G/14 for ImageNet has 1843 M parameters (Simonyan and Zisserman 2015)). The acceleration of DNN is limited by the massive fetches of the synaptic weights. Thus, from the algorithms' point of view, a large memory capacity is preferred for reducing the expensive data transfer from the on-chip/on-chip buffer. In the meantime, thanks to the CMOS technology scaling and innovations on emerging nonvolatile memories (eNVMs), on-chip memory capacity is also increasing rapidly (e.g., 256 Mb SRAM (Song et al. 2018), 8 Mb RRAM (Kawahara et al. 2012)). Accordingly, researchers are developing CIM architectures with on-chip embedded memories such as SRAM (e.g., CIMAT (Jiang et al. 2020a)) and resistive random-access memory (RRAM) (e.g., PRIME (Chi et al. 2016), PipeLayer (Song et al. 2017)).

This chapter aims to present state-of-the-art mixed-signal CIM designs from architecture techniques such as network mapping and pipeline design to hardware implementations, including device exploration and circuit techniques, hoping to inspire the research community for future interdisciplinary collaborations on this exciting research topic. Most techniques to be discussed here are proposed for

DNN-based algorithms but can be extended to other machine learning algorithms with similar computing principles. It should be noted that CIM could be implemented in the fully digital domain (Wang et al. 2019) or fully analog domain (Chang et al. 2019), with the mixed-signal mode being the most common manner with a balance between efficiency and scalability. Thus, this chapter is focused on mixed-signal CIM designs.

DNN Basics and Corresponding CIM Principle

DNNs are a family of machine learning algorithms that mimic the principle of the human brain. Generally, they are presented as a network of interconnected nodes called neurons. The connections between neurons are called weights. Typically, neurons are aggregated into layers: an input layer, an output layer, and one or more hidden layers. Weights are learnable parameters controlling the strength of the connections between neurons. Figure 1a shows a generic structure of one layer in NN. In recent years, the remarkable success of the DL algorithms is mainly promoted by the DNNs on image classification, which expands its power and effectiveness across a wide range of applications such as natural language processing and autonomous driving. DNN on image classification is taken as a sample problem in this chapter.

The weights of a DNN are generally initialized with random numbers and learned through training. Once the model achieved desirable performance by training, it could be used for inference tasks. The most widely used training method for DNN today is stochastic gradient descent (SGD). The training process mentioned in this chapter is based on this method.

Figure 1b presents the basic diagram of the DNN training process. In general, the process of CNN could be divided into four steps, namely, (1) inference/feed-forward (FF), (2) error calculation (EC), (3) gradient calculation (GC), and

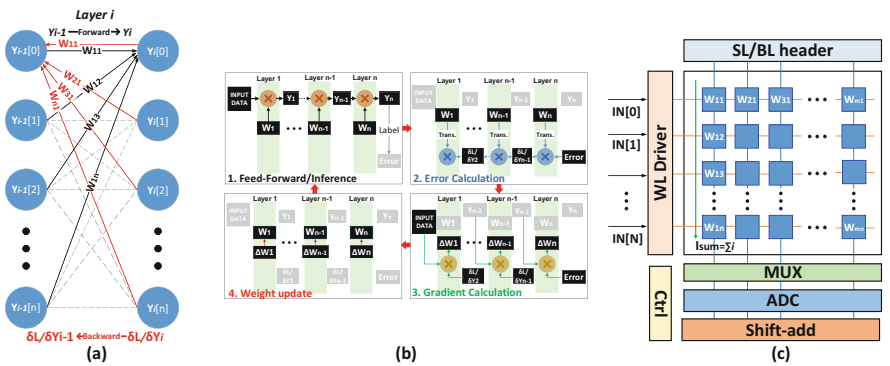


Fig. 1 (**a**) Generic structure of one layer in DNN. (**b**) Basic diagram of the CNN training process. (**c**) Mixed-signal MAC operation in one memory sub-array

(4) weight update (WU). These four steps run in a loop to obtain a well-trained model through iterations. In the FF/inference step, the input is processed layer by layer and finally generates the output as desired. In the classification task, the final output is normally a distribution that indicates the class that the input belongs to. For the inference task, this distribution will be directly used to decide the label. For the training case, this distribution will be used to calculate a loss with respect to the ground truth label, which indicates how far the predicted output is away from the desired value. During the FF process for each layer, the basic operation is the convolution between the input and weight followed by neuron activation, as shown in Eq. 1.

$$\boldsymbol{Y_n} = f\left(\boldsymbol{Y_{n-1}} * \boldsymbol{W_n} + b_n\right) \tag{1}$$

where Y_n is the current layer's output, usually regarded as activations to the next layer due to the layer-by-layer procedure. b_n is the bias of the layer and could be eliminated in some cases. $f(\cdot)$ denotes the activation function such as sigmoid, ReLU, etc. Other post-processing functions, such as pooling and batch normalization applied to the output, are not shown in the equation. During the EC process, the main goal is to calculate the gradient of inputs of each layer with respect to the loss ($\partial L/\partial Y_n$), which is also called the error. A method called chain-rule is used to calculate the gradient layer by layer from the back to the front. The operation of the EC process is similar to FF except that the weight matrix $\left(W_{n+1}^T\right)$ needs to be transposed for backward propagation. The EC could be represented as Eq. 2.

$$\frac{\partial L}{\partial \boldsymbol{Y_n}} = \frac{\partial L}{\partial \boldsymbol{Y_{n+1}}} \frac{\partial \boldsymbol{Y_{n+1}}}{\partial \boldsymbol{Y_n}} = \frac{\partial L}{\partial \boldsymbol{Y_{n+1}}} * \boldsymbol{W_{n+1}^T} \tag{2}$$

Then the weight gradient $\frac{\partial L}{\partial W_n}$ is obtained by another convolution between the activation and error, as shown in Eq. 3. Finally, the weights of the current layer are updated by $-\frac{\partial L}{\partial W_n}$ modulated by the learning rate (LR), which is also called ΔW_n, as shown in Eq. 4.

$$\frac{\partial L}{\partial \boldsymbol{W_n}} = \frac{\partial L}{\partial \boldsymbol{Y_n}} \frac{\partial \boldsymbol{Y_n}}{\partial \boldsymbol{W_n}} = \frac{\partial L}{\partial \boldsymbol{Y_n}} * \boldsymbol{Y_{n-1}} \tag{3}$$

$$\boldsymbol{W_n}(t) = \boldsymbol{W_n}(t-1) - lr \cdot \frac{\partial L}{\partial \boldsymbol{W_n}} = \boldsymbol{W_n}(t-1) + \Delta \boldsymbol{W_n} \tag{4}$$

Massive convolutions during CNN processing can be essentially mapped to 2D vector-matrix multiplications (VMMs) between the input vector and weight matrix. The CIM crossbar array as shown in Fig. 1c, which performs the multiply-and-accumulate (MAC) operation with perpendicular input rows and output columns, can naturally support VMMs efficiently. The flattened 2D weight matrices can be stored on CIM arrays and weight value can be mapped to memory cells as stored charge, device conductance, or other forms. The memory cell is represented

by the blue box, which could store binary or multi-bit weight in theory. The multiplication is done in an analog fashion in which the input vector is loaded in parallel as voltage to the rows and multiplied by weight conductance to generate products in the form of current. Then current summation along columns represents the final MAC output. Analog-to-digital converter (ADC) is normally employed to quantize the analog MAC output to binary bits for further digital processing (e.g., shift-and-add, activation function, and pooling). Thus the CIM is essentially a mixed-signal computing scheme. Theoretically, VMMs could be performed in CIM in a fully parallel fashion if assuming all the rows and all the columns can work simultaneously. In reality, usually only a part of rows/columns could be synchronously turned on due to limited ADC resolution or the mismatch between the column pitch and the peripheries.

Architecture and Algorithm Techniques for CIM

Compared to conventional computer architecture design, the development of CIM design necessitates more cross-layer optimizations from architecture and algorithm levels to circuit and device levels. A clever algorithm or novel architecture can significantly relax the device requirements while encouraging device properties can help improve the system's efficiency. This session will mainly concentrate on CIM design considerations and techniques on the architecture and algorithm level, including a general hierarchical architecture of CIM, network mapping strategies, pipeline design, and quantization techniques for CIM. Some of those techniques are highly related to device/circuit designs. Hardware implementations will be discussed in-depth in the next session.

Hierarchical Architecture of CIM

Compared to the single-core processor, which needs to employ complex circuits to increase pipeline parallelism in sequential programs, the tiled multi-core architecture can easily improve throughput by hiring more cores for parallel workloads (Keckler et al. 2009). Considering the parallel nature of DNN computation, most CIM architectures are based on tiled architecture. Figure 2a shows the top-level diagram of a representative tiled CIM architecture, mainly composed of tiles (labeled T), essential functional blocks for NN computation, global control, and buffer connected with on-chip interconnections. Essential functional blocks generally include accumulation units, pooling, activation units, etc. The accumulation units are needed to sum up the partial sums from multiple tiles. The summed-up results will then be sent to activation units (or/and pooling units if necessary). Finally, the activations will be sent back to the global buffer. The global control will schedule the new feature map to other tiles to operate the following computations. Additional function blocks such as de-pooling, find_max, and random number generation should be included to support on-chip training. As shown in Fig. 2b,

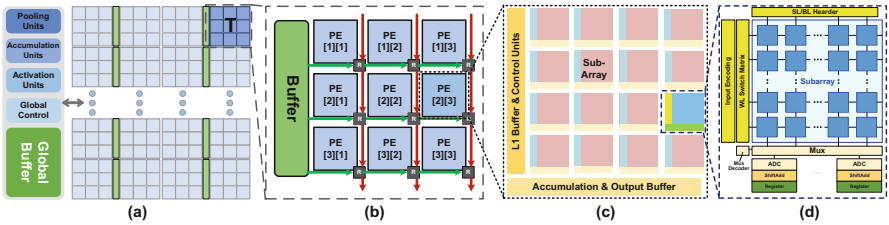


Fig. 2 Hierarchical architecture of CIM. (**a**) Chip-level structure composed of tiles. (**b**) Tile-level structure composed of PE. (**c**) PE-level structure composed of sub-arrays. (**d**) Sub-array structure

a tile usually contains multiple in-situ processing units (PEs) and input/output buffers connected with routers. The routers make it possible to communicate among PEs and transfer partial sums from PEs to the top level. Figure 2c shows a PE structure that contains one or a few CIM subarrays, local input/output buffer, and accumulation units if necessary. The intra-PE accumulation units are normally used to sum up partial sums from sub-arrays. Figure 2d shows a typical CIM sub-array structure consisting of a crossbar memory array and compute peripheries including input encoder (DACs), WL switch matrix, ADCs, shift-adders, and registers. Especially, DACs/ADCs provide the scalability and flexibility for the mixed-signal communication between the sub-arrays and upper level. To summarize, CIM design is usually based on a multi-core hierarchical architecture (Jiang et al. 2020b), in which elementary MAC operation is performed in analog domain at array level while further processing such as activation function and accumulation is implemented by digital.

Network Mapping Strategies

The process of the convolutional computation is shown in Fig. 3: in layer <n>, the size of input feature maps (IFMs, namely activations of layer <n> in the FF process) is $H \times W \times C_{in}$ (where H/W is the IFM plane height/width), which are the outputs from layer <n-1>. Here, IFM and a corresponding output feature map (OFM) are used in place of activations to denote the input/out of the convolution operation only. The activation function is not included here, so it is not precise to call the output activations. The size of each 3D kernel is $K \times K \times C_{in}$ (C_{in} is the number of IFMs/input filter channels) with kernel depth of C_{out} (i.e., there are C_{out} such 3D kernels). Thus the total size of the kernels in the layer will be $K \times K \times C_{in} \times C_{out}$. To get the outputs, a group of IFMs (with size $K \times K \times C_{in}$) will be selected at each time and to be multiplied and accumulated with C_{out} kernels with size $K \times K \times C_{in}$, then each of them will generate a $1 \times 1 \times 1$ output. The output from the top kernel (shown as the light orange cube) goes to the front, and the output from the bottom kernel (shown as the dark orange cube) goes to the back. Thus, in total, there will be $1 \times 1 \times C_{out}$ outputs. As shown in Fig. 3, it could be considered that the

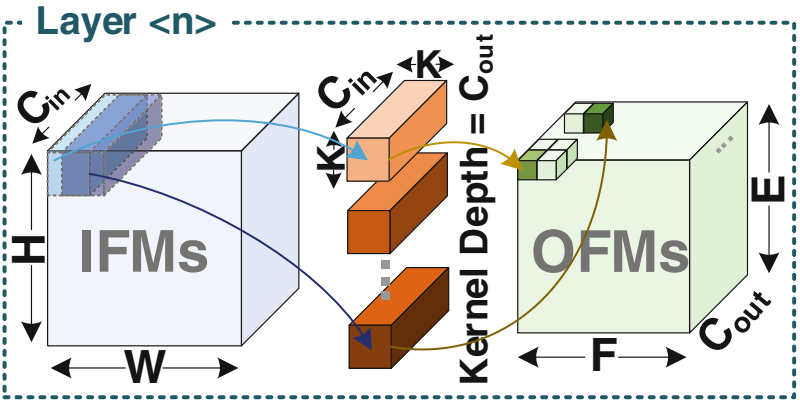


Fig. 3 Computation process in a convolutional layer of DNN

kernels are "sliding over" the IFMs, and perform elementwise multiplications with a certain stride. Then the products of each elementwise multiplication in each 3D kernel will be summed up to get the final outputs. The size of output feature maps (OFMs, namely outputs of layer <n>) will be $E \times F \times C_{out}$ (E/F is the OFM plane height/width, which depends on IFM size and stride number). Besides convolutional layers, most CNN also has several fully connected layers, which could be viewed as a special case of DNN with kernel size $1 \times 1 \times C_{in} \times C_{out}$, IFM size $1 \times 1 \times C_{in}$, and IFM size $1 \times 1 \times C_{out}$.

To correctly perform inference and training processes on a CIM architecture, network mapping, especially weight mapping, is a significant part of hardware implementation. The weight mapping strategies for the CIM architecture can be divided into two parts: mapping methods for inference and mapping methods for training.

Mapping Methods for Inference

In the CIM architectures for DNN computations, the weights need to be mapped into crossbar arrays as the conductance of each memory cell. In such a way, the 3D VMM multiplication could be transformed into a 2D dot-product multiplication. Then all the dot products from the same 3D kernel are summed up to get the final output.

Considering the nature of the crossbar array, the straightforward mapping method is to unroll each 3D kernel into a long column (i.e., kernel-flatten mapping). Products from each cell will be summed up along columns, representing the final output. In this way, all the kernels in one convolutional layer are converted to a large weight matrix. Figure 4a shows an example of this straightforward weight mapping method (Gokmen et al. 2017). One 3D kernel with size $K \times K \times C_{in}$ could be unrolled to a long column whose length equals to $K \times K \times C_{in}$, and with kernel depth equals to C_{out}, there are C_{out} columns in total. As a result, the 3D kernels could be mapped into a large weight matrix whose length and width are

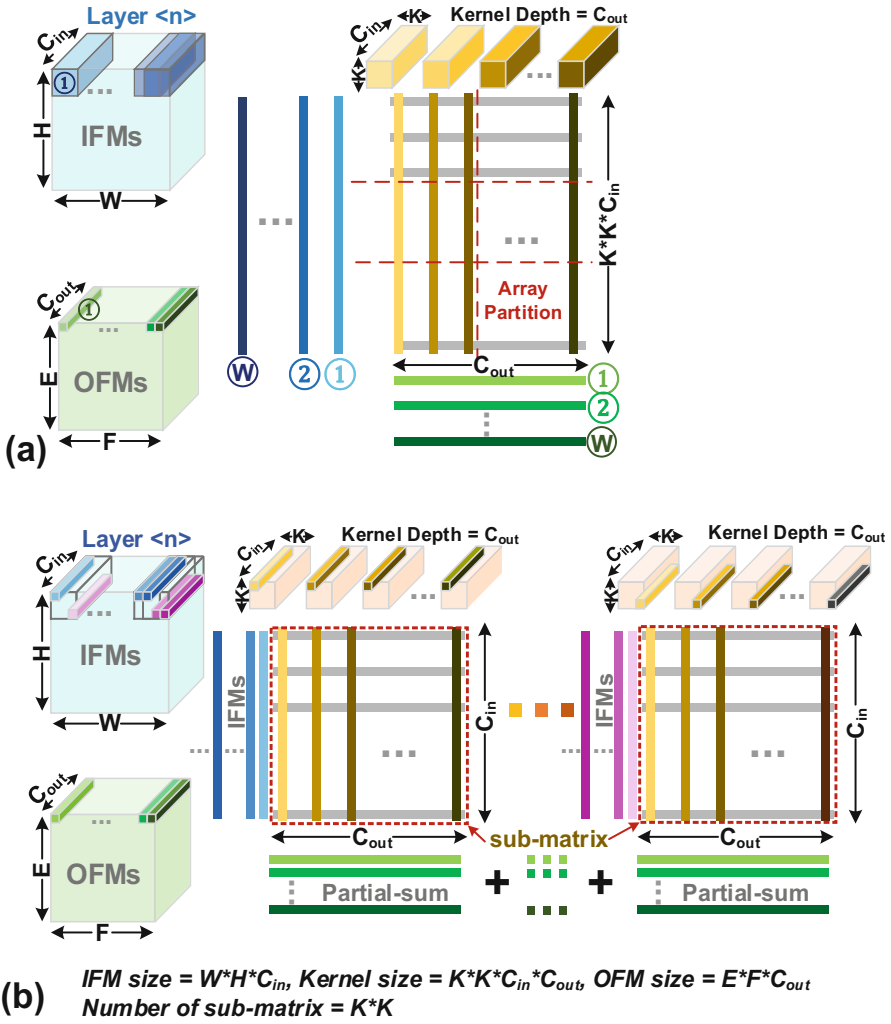


Fig. 4 (**a**) Kernel-flatten mapping. (**b**) Kernel-splitting mapping

equal to $K \times K \times C_{in}$ and C_{out}, respectively. To practically map large convolutional layers on-chip, array partitioning is necessary. A single large matrix could be cut into several sub-arrays and parallelize the computation efficiently. Because of the convolution manner, in each computation, a part of IFMs multiplies with each kernel to obtain one output of OFM. To generate the total OFMs in one layer, the kernels must slide over the IFMs many times. It can be observed that during the "kernel sliding," part of the input data will be reused for the computation of the next output. Considering the massive dot-product operations in convolutional layers, these frequent reloads of input data may cause significant energy and latency

waste. In a typical CNN structure, the kernel size normally varies across different layers. Thus the unrolled weight matrix size for different layers will be quite different, which leads to a various number of sub-arrays to be used to represent different layers. With the kernel-flatten mapping method, it is impractical to reuse the unrolled input data among a various number of sub-arrays since the design of interconnects and control circuits will be complicated and non-reusable.

An alternative mapping method (Peng et al. 2020) is proposed to maximize the input data reuse as shown in Fig. 4b, which could be named as kernel-splitting mapping. Unlike the kernel-flatten mapping method, where all the 3D kernels are unrolled into a large matrix, the weights at different spatial locations of each kernel are mapped into different sub-matrices. The group (sub-matrix) of these partitioned data is sorted according to the spatial location of partitioned data in each kernel. For example, all the partitioned data located at the left-top channel at each kernel will be reorganized as one group. Then it will be implemented into one sub-matrix, and the height and width of each sub-matrix should be equal to $1 \times 1 \times C_{in}$ and C_{out}. Hence, K $\times$ K sub-matrices are needed for all the kernels. Similarly, the size of such a sub-matrix could also be large. In this case, each sub-matrix can be represented by a group of sub-arrays, defined as a PE. Based on this kernel-splitting mapping method, which cuts the kernels into several PEs according to their spatial locations and assigns the input data into corresponding ones, it is possible to reuse the IFMs among these PEs efficiently.

Figure 5 shows an example of kernel-splitting mapping and processing a convolutional layer with a 3×3 kernel. Thus, nine processing units (PEs) are hired correspondingly and each PE consists of several CIM sub-arrays. Firstly, at the first cycle (i.e., T = 1), all the input data are assigned to the corresponding PE. For example, an input vector with a length C_{in} (i.e. IFM[1][1]) is assigned to PE[1][1], similarly IFM[1][2] is assigned to PE[1][2] and IFM[1][3] is assigned to PE[1][3]. After first-cycle computations, partial sums of size $1 \times 1 \times C_{out}$ from these nine PEs will be summed up to get the final OFM. Then, at the next cycle (i.e., T = 2), the

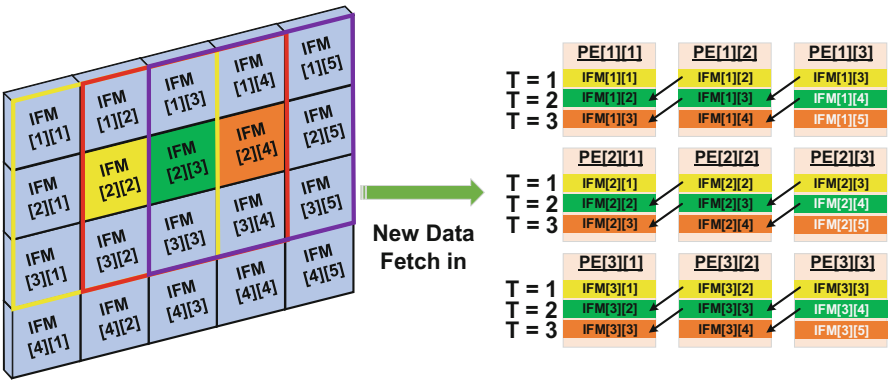


Fig. 5 An example of IFM reuse with kernel-splitting method

IFMs used for the next computation are transferred to the neighboring PEs, and the useless IFMs will be released. For example, IFM[1][2] is transferred from PE[1][2] to PE[1][1] and IFM[1][3] is transferred from PE[1][3] to PE[1][2], while IFM[1][1] is unloaded (will not be used anymore). As the example shows, with such novel data flow, only one-third of input data are newly introduced from buffers, and two-thirds of them could be reused from the neighbor PEs. Thus, by passing the used IFMs in the same direction as the kernel slides over the inputs, the IFMs can be reused efficiently. For general cases, with kernel size $K \times K$, and stride equals to S, only S/K of required input data are newly transferred in each cycle and the rest $(K - S)/K$ of input data can be fetched from neighboring arrays.

Mapping Method for Training

As aforementioned, for the error calculation (EC) process in the CNN training, CIM arrays need to perform convolutional computation with transposed weight matrices. Although both mapping methods for inference can be used for transposing computation during training, the kernel-flatten mapping will make the forward and backward operation unbalanced since the column output in FF is directly the entire partial sum. In contrast, the row output in error calculation is just part of the partial sum. Considering the computing balance between FF and EC process, a kernel-splitting mapping scheme is preferred in transpose CIM architecture. For the FF, the product of the input sliding window with the same kernel across all the channels is summed up to obtain one output of OFM, which means all dot products in the same column are summed up. However, for the EC, the product of the input sliding window and the same channel across different kernels are summed up to obtain one $\partial L/\partial Y_n$ output, which means all dot-products in the same row need to be summed up. In other words, the transposed weight matrix is essential for error calculation. As shown in Fig. 6, W_{n+1} is the weight matrix in FF while W_{n+1}^T is the transposed weight matrix for EC. They can be mapped to the same memory array. In the EC, the input vector (i.e., the error of the deeper layer $\partial L/\partial Y_{n+1}$) is applied to the column in parallel, and the output vector (i.e., the error of the current layer $\partial L/\partial Y_n$) is obtained from the row in parallel. The inference and error calculation process can be performed within the same CIM processing unit with such a transpose mapping method. Consequently, no additional memory storage or access is needed for the transpose weight matrix. It should be noted that the bidirectional read capability of the device is demanded to support such transpose mapping strategy.

Number Representation in CIM Architecture

The previous section discussed the strategies for mapping the network to the hardware from the spatial position's point of view. However, how the number is represented in the hardware will also matter for correct calculation by the hardware. The 2's complement representation is the most common method for signed number calculations in the digital system, and the sign extension of the operands is usually required to guarantee the correctness of 2's complement operation. For the multiplication between weight and input in CIM, the sign extension of the weight means more cells are needed for one weight representation, and the sign extension of

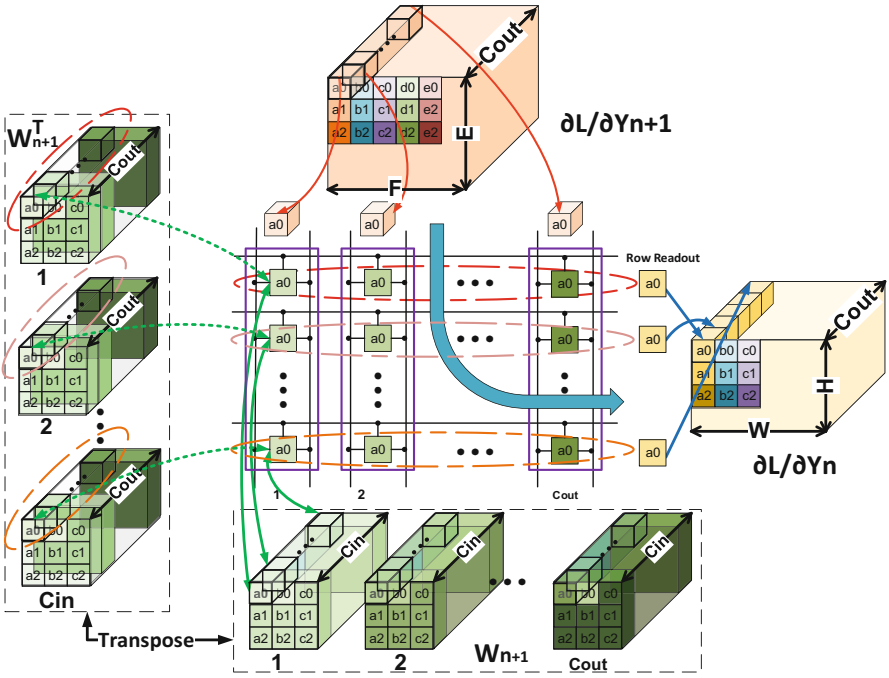


Fig. 6 An example of transpose weight mapping that implements a0 element in a 3 × 3 filter. (With permission of Jiang et al. 2020a)

the input means more cycles are used for one MAC. Thus, directly applying the sign extension scheme to CIM will significantly increase the area, power consumption, and latency cost. Instead, the MAC operations of 2's complement values could be done by first accumulating unsigned bit sequences and then multiplying the signed scale. This is because the 2's complement representation could be viewed as the weighted sum of unsigned binary weight with signed bases (Eq. 5). Thus, in some binary-cell-based work (Jiang et al. 2020b), the binary weight sequence is first treated as the unsigned weighted sum in the memory array and then multiplied back the scale and sign information (by shift-add and/or inverse function) to the 2's complement representation. Moreover, this representation could be extended to the case where the memory cell is more than binary. For example, the 2's complement representation of the 2-bit per cell case is shown in Eq. 6. In Eq. 5, b_i is binary value. Thus, for the sum of the adjacent two values $b_i \cdot 2^i + b_{i-1} \cdot 2^{i-1}$, it could be viewed as $b_i \cdot 2 \cdot 2^{i-1} + b_{i-1} \cdot 2^{i-1}$, which could further be grouped to $(b_i \cdot 2 + b_{i-1}) \cdot 2^{i-1}$. The value in the bracket could be directly represented as a 2-bit value $b_i b_{i-1}$ as shown in Eq. 6. In CIM array, such 2-bit value can be stored in one multi-bit cell (i.e., 2-bit per cell in this case). It should be noticed that the most significant bit (MSB) with the negative base could not be combined with positive bases as the other bits (b_n and b_{n-1} cannot be combined). A simple solution is to use one additional cell

to store the MSB only. Thus, if one wants to use 2-bit per cell to represent an 8-bit weight, 5 (= 4 + 1) cells are needed. Compared to direct extending the sign bit, this two-step calculation introduces less overhead, which is also applicable for inputs for two reasons. First, in today's DNN, ReLU is generally used as the activation function, making the inputs always positive (in inference task), which means MSB is always "0". In this case, the MSB could be skipped. Second, for a more general case, where negative inputs are also possible (e.g., in the EC step of training), only an extra cycle is needed to process the MSB, which will not contribute to a big portion of energy and latency overhead while usually multiple cycles are needed with direct sign extension.

$$x = b_n \cdot \left(-2^n\right) + b_{n-1} \cdot 2^{n-1} + \cdots b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0 \quad (5)$$

$$x = b_n \cdot \left(-2^n\right) + b_{n-1} \cdot 2^{n-1} + \cdots b_3 b_2 \cdot 2^2 + b_1 b_0 \cdot 2^0 \quad (6)$$

Beyond the 2's complement representation for signed weights, there are mainly three methods used in the reported CIM designs: (1) implement the MAC independently with positive and negative weights; (2) implement the MAC with positive weights and use the reference column to move them back to zero-centered weight; (3) use a differential pair of cells per weight to represent the signed weight value.

The first method uses two copies of the weight matrix: one with positive weights only and the other has negative weights only. Then, both the matrices are calculated in an unsigned manner. No sign extension is needed. Then the result from the negative arrays will be subtracted from its corresponding positive ones using pure digital circuits. A simple illustration of this method is shown in Fig. 7a. The disadvantage of this method is that it needs twice the hardware resources to represent one weight matrix. Also, it is only suitable for inference implementation as the training could make the weight change from positive to negative and vice versa. The advantage of this method is that the two resulting matrices are sparser than the original one, which may release the design complexity of the periphery circuits.

The second method relies on the single reference column for an array, which is one of the most common methods used for multi-level cells and could be applied for both training and inference. The basic idea is shown in Eq. 7. For real implementation, the weights in the kernel are limited in a finite range. Assume weight $w_i \in [-b, b]$, thus the shifted weight $w_i + b \in [0, 2b]$, which is always positive. This all-positive weight matrix will be mapped to the memory array, and then the MAC could be done with unsigned weights. A reference column will be attached to the array to calculate the second term $\sum_i in_i \cdot b$, as shown in Fig. 7b. Since b is a constant decided by the weight range, the reference column could be shared by the whole array. Finally, the shift back could be done in the analog domain before ADC or by the digital circuits after ADC. The former is more difficult to implement by circuits as the subtraction in the analog domain is not as easy as the addition. On the other hand, the latter will introduce more errors since both terms have quantization errors introduced by the ADC.

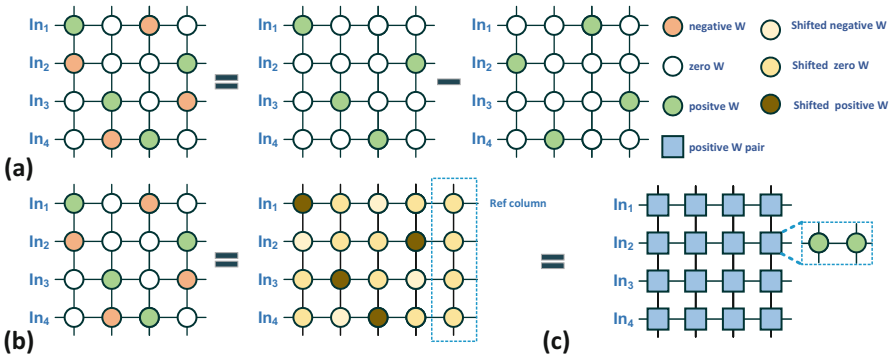


Fig. 7 Weight number representation: (**a**) positive weight matrix + negative weight matrix; (**b**) shifted weight matrix + reference column; (**c**) weight matrix with two differential cells representing one weight

$$\begin{aligned} partial\ sum &= \sum_i in_i \cdot w_i \\ &= \sum_i in_i \cdot [(w_i + b) - b] \\ &= \sum_i in_i \cdot (w_i + b) - \sum_i in_i \cdot b \end{aligned} \tag{7}$$

The last method is based on differential pair of two cells, which could be described by Eq. 8 and Fig. 7c. Similar to the second method, the subtraction could be done before summation in an analog manner (Liu et al. 2020) or after summation using digital circuits based on the implementation. This method doubles the cells needed to represent one weight. While the first method also needs double the cell number, this method does not guarantee a similar sparsity deduction as the first one. Compared to the second method, which has a common shared reference, this method introduces more overhead. However, this method employs two adjacent cells in a differential manner, and it has more flexibility to cancel the local process variations and fine-tune the weight more precisely.

$$\begin{aligned} partial\ sum &= \sum_i in_i \cdot w_i \\ &= \sum_i in_i \cdot \left[w_i^+ - w_i^-\right] \\ &= \sum_i in_i \cdot w_i^+ - \sum_i in_i \cdot w_i^- \end{aligned} \tag{8}$$

Pipeline Design in CIM Architecture

After mapping the network to the hardware, techniques on dataflow are usually used to help improve the DNN acceleration in the CIM architectures. The pipeline is a

widely used technique in today's computer architectures. The layer-by-layer process nature of CNN makes it very natural to employ a pipeline for its implementation. In the recent CIM architecture, intra-layer or inter-layer pipeline design has been considered to improve throughput and lower design costs on hardware.

Intra-Layer Pipeline

As one of the representative CIM architectures, ISAAC (Shafiee et al. 2016) attempts to improve the throughput of the architecture by using an intra-layer pipeline, which first computes a small portion of one layer and then assigns the partial outputs as the input for the proceeding layer in the next cycle. Figure 8 shows an example of a buffer requirement for such a pipeline, assuming that a 6×6 input feature map is being convolved with a 2×2 kernel. As shown in Fig. 8a, the generated outputs 0, 1, 2, . . . , 6, 7 (in green) from the previous layer <n – 1> are placed in the input buffer for layer <n>. At this moment, enough information has been acquired to start the operations for layer <n>. So the first output for layer <n> could be produced by stored inputs 0, 1, 6, 7 (in red box). Then, after layer <n − 1> produces output 8, layer <n> can start to calculate the next output with input 1, 2, 7, 8 (Fig. 8b). In such a way, every new output produced by layer <n-1> triggers the pipelining of layer <n>, which could perform kernel-based operations step-by-step. Figure 8c shows the status of the input buffer after a few steps. At the same time, serviced inputs (in gray) can be released from the input buffer. ISAAC's intra-layer pipeline enables a saving in the buffering requirements between successive layers and a throughput improvement. However, the pooling function in DNNs could rapidly shrink the layer size, limiting the intra-layer pipeline efficiency. Another drawback is increasing the instant power as all the layers are expected to be active simultaneously, which may exceed the power budget for the edge devices.

Inter-Layer Pipeline

Since CNN computations follow a layer-by-layer process, a possible inter-layer pipeline could significantly speed up the whole process. Specifically for training, both the FF and EC processes are executed in a layer-by-layer fashion. CIMAT (Jiang et al. 2020a) proposes an inter-layer pipeline design, which allows the new input to enter the pipeline every cycle within a batch. A pipeline dataflow for training ResNet-18 is shown as an example in Fig. 9a. Since layer size varies a lot in one network, sometimes one pipeline stage needs to include several layers to match

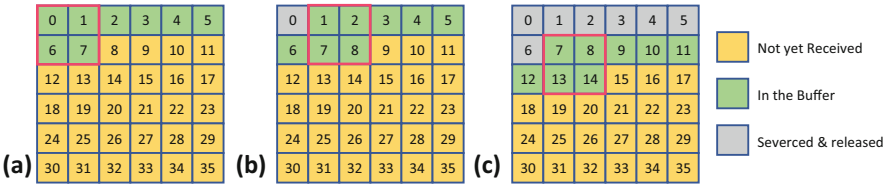


Fig. 8 Input buffer requirement for the intra-layer pipeline

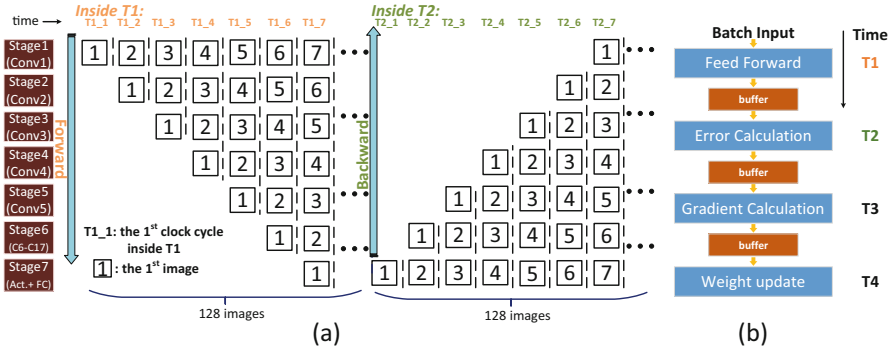


Fig. 9 (**a**) An inter-layer pipeline example inside FF and EC for ResNet-18. (**b**) Training process in timeline of 7 T SRAM-based CIM architecture

the latency of each stage. In this example for ResNet-18, the latency of layer 1, 2, 3, 4, 5 to process an entire image is almost the same, approximately equal to the total latency of the sixth to 17th convolution layers. Therefore, Layers 1, 2, … 5 are treated as pipeline stage 1 to stage 5, respectively, while layer 6 to layer 17 are grouped as one stage (stage 6). Stage 7 consists of a fully connected layer and other activation function blocks. Starting from FF computation, in cycle T1_1, the first input (i.e., first image) enters stage 1, which performs the FF computation. At the end of T1, the outputs need to be sent to the buffer or off-chip memory for future EC computation. Then at T1_2, the second image enters stage 1 while the first image is passed to stage 2. In such a way, except for some initial cycles, all the stages are occupied simultaneously in the pipeline. As described in Fig. 9, data dependency exists in FF and EC computations. EC process can only start after obtaining the last layer's error outputs, indicating the FF process is completed. The pipeline dataflow of the EC and FF processes is highly alike but in opposite directions. Figure 9b shows the entire training process in the timeline. First, one batch of images is fed into the CIM architecture stage by stage in the forward direction for the FF process. After finishing the FF process of one batch (T1), the EC process starts to operate stage by stage in the backward direction (T2). The generated intermediate data in FF and EC process needs to be saved in buffer or off-chip memory for gradient calculation. For the gradient calculation (GC) process, after the errors are obtained for one batch, they are used together with the activations to calculate the weight gradients (T3). The GC is performed after the batch FF and EC process. Finally, weight gradients are averaged across the batch to update the weights (T4). Pipelayer (Song et al. 2017) shows a similar inter-layer pipeline design hiring another set of weight matrices, activating the inter-layer pipeline between the FF and EC processes. The reason is that, with two copies of the weight matrix, FF and EC processes can be executed simultaneously on different CIM arrays. The drawback will be the doubled hardware cost. There is an improved CIMAT architecture based on a novel 8 T transposable SRAM design (Jiang et al. 2020a),

which can perform FF and EC on the same array concurrently without employing additional CIM arrays. The proposed 8 T SRAM bit-cell can perform bidirectional read simultaneously (described in the section "Device Technologies"), which means the CIM sub-array equipped with such 8 T SRAM is able to support bidirectional MAC calculation synchronously. Thus, an aggressive inter-layer pipeline design is proposed based on this ability, as shown in Fig. 10a. The stage configuration of the FF and EC process is the same as the pipeline design in Fig. 9. However, instead of waiting for the completion of the FF process, the EC process could form pipelines together with the FF process to increase throughput significantly. In addition, as long as the activation and error outputs of the first image are ready, the weight gradient calculation could start to work. If the latency of the GC stage approximately equals the latency of the FF/EC stage, gradient calculation could also work in a pipeline fashion together with FF and BP processes. The hardware resource highly decides the latency of the GC stage since it could be processed in parallel. Thus it could be controlled to match the stage latency of FF/EC. The state of each stage as a function of time is shown in Fig. 10b. For example, at T14, FF/EC stage 7 (FF/EC S7) is performing the FF of image 8 and the EC of image 7 simultaneously. Meanwhile, GC stage 1 (GC S1) is able to calculate the weight gradient for image6 because necessary activations and errors of image 6 have been obtained in the T12 cycle and the T13 cycle, respectively. Such a fully inter-layer pipeline design could further improve the throughput of the training process with a small hardware overhead.

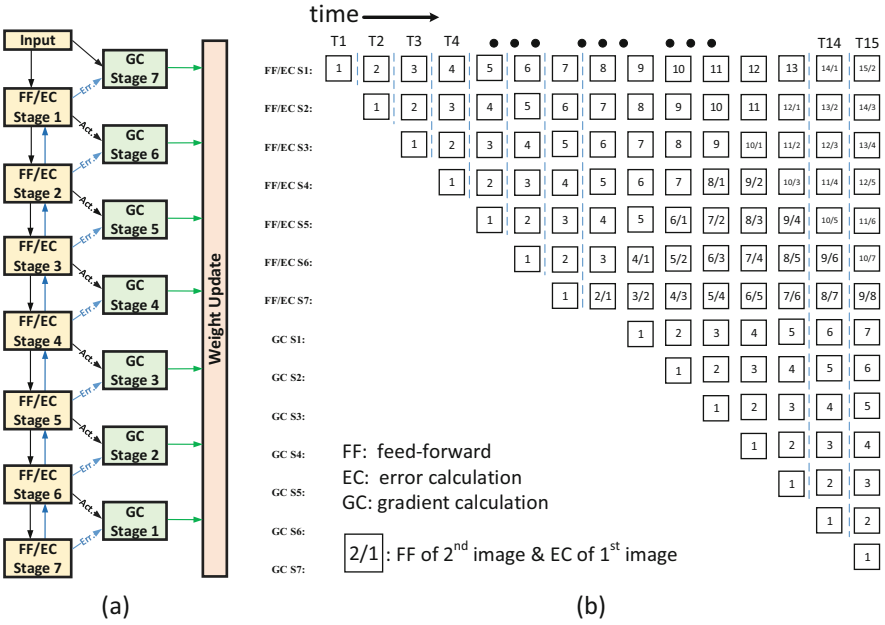


Fig. 10 (**a**) Dataflow of an improved inter-layer pipeline using 8 T SRAM-based CIM architecture. (**b**) The state of each stage as a function of time

Besides, the optimized pipeline is also beneficial to energy saving due to the reduced off-chip memory access and standby leakage of circuits. The overhead of such highly pipelined architecture will be the requirement of the large on-chip buffer capacity to execute the pipeline.

Quantization Techniques in CIM Architectures

In general-purpose platforms such as GPU, the training and inference for DNN are usually run in 32-bit or 64-bit floating-point number representation. On one side, the higher the number precision, the higher the energy will be consumed per operation. On the other side, the floating-point number calculation usually requires more hardware resources and consumes more power than the fixed-point number calculation. Therefore, for the DNN accelerators, especially for edge devices, quantized training and inference are usually utilized to minimize the chip area and power consumption.

The quantized DNN algorithm is also an important and active research topic for CIM architecture. Although CIM has the potential to conduct pure analog calculation, which means both input and weight could be high precision theoretically, the fact is that the dynamic range of the outputs from the circuits/devices is limited. In general, inputs will be limited to $1 \sim 2$ bits per cycle, while cells are limited to $1 \sim 5$ bits presentation according to the devices. High-precision inputs could be applied to the array cycle by cycle, while high precision weights could be represented by several cells per weight. In this case, shift-adders are essential to accumulate the corresponding partial sums to get the final high-precision MAC outputs. Obviously, the lower the input/weight precision, the lower the hardware cost to implement the model with the CIM architecture. In addition, it is tough to implement floating-point operation in CIM since the weights are saved stationary in memory, which is difficult to realize radix points alignment. Consequently, most of the proposed CIM architectures are designed for fixed-point operation inside the arrays. This chapter focuses on fixed-point calculation in CIM. Recently, some CIM designs supporting floating-point calculation were proposed (Imani et al. 2019), and the readers could check the reference for the floating-point case if interested.

The quantization of DNN has been studied ever since the 1990s (Presley and Haggard 1994). According to the targeted operation modes, it could be divided into two categories: low precision inference and low precision training. The quantization for inference aims to obtain a model that mainly includes low-precision weights and activations in the inference stage. Without specific techniques applied, directly quantizing the weights and activations of a well-trained floating-point model to 8-bit fixed-point inference is usually possible to have negligible performance loss for most image classification tasks (Vanhoucke et al. 2011). Low precision training targets reducing energy costs and required resources for training/quantization. It is much more challenging to guarantee comparable performance with the floating-point network with low-precision training. As the weights need to be updated during training, which may cause large dynamic range changes for both weights and activations, using a low precision fixed-point number representation across all

the training processes is risky. The safer choice is to use at least 16-bit floating-point numbers for low-precision training, especially for those complex datasets requiring training from scratch.

Several popular quantization techniques that are widely used in CIM architectures will be briefly introduced, which are: (a) dynamic fixed-point quantization; (b) mix-precision weight; (c) stochastic quantization. More aggressive quantization algorithms that are not applicable to CIM are not covered in detail in this chapter.

The dynamic fixed point quantization (Courbariaux et al. 2015) is proposed as a compromise between fixed-point and floating-point for training tested on MNIST and CIFAR-10 datasets and successfully reduced the precision to 10 bits for FF and EC. The dynamic fixed-point will have a scaler parameter to scale a fixed-point number as the exponential part of the floating-point number called the shared exponent. This shared exponent will change during training while making the MAC operation stay in low fixed-point precision. For example, the shared exponents could be the moving average of the maximum value of the numbers participating in MAC. A DNN model could have several shared exponents for groups of numbers, such as one per layer or one per channel. The original paper only explores its effectiveness on moderate tasks such as CIFAR-10 classification, but this concept is widely used in the following proposed quantization algorithms.

The mix-precision weight is another technique used in Courbariaux et al. (2015) and many other quantization algorithms based on the observation that weights used in FF and EC could be in lower precision than in WU (for gradient accumulation). Thus during training, a copy of high-precision weights will be stored in memory. For the FF and EC, this copy will be quantized to lower precision for economic convolution calculation. After the weight gradients are calculated, they are used to update the high precision weights version instead of the quantized version for FF and EC calculation.

Stochastic quantization is introduced in Hubara et al. (2016) for gradient quantization to avoid the underflow of small gradients. In stochastic quantization, the gradient is used as a probability to round to its nearest quantized levels. Compared to deterministic quantization, nonzero probabilities are assigned even to very small gradients. Stochastic quantization could make the gradients effective even with lower precision, which means the weight used in the weight update could be held in lower precision.

Quantization techniques (a) and (b) are widely used to train a quantized model for inference-only mode. For the dynamic fixed-point value, the shared exponents are fixed during the inference stage once the training is done. Thus the model could be mapped to the hardware using the fixed-point format. Since the training overhead is not considered, the high precision copy of the weight could be implemented even in floating-point, which means the gradient could be floating-point. Also, since EC is not a part of inference, errors could be implemented in high precision floating-point to reduce the noise caused by quantization during training. Compared to quantizing the floating-point model directly after training, these quantization-aware training techniques could achieve lower precision inference for similar accuracy on the same model structure.

During the training, all three techniques are important to keep the overall on-chip hardware overhead small. Activation, weight, and error are all in dynamic fixed-point format to support low precision fixed-point MAC for both FF and EC. Weight precision is low due to the mixed precision, which will benefit FF and EC since it takes part in both. Stochastic quantization is important to keep the gradient at low precision, thus decreasing the required weight precision during weight update.

Some modern quantization algorithms achieve extreme quantization with one or more of these three techniques, thereby becoming attractive for CIM implementation. The DoReFa network (Zhou et al. 2016), which utilizes all three aforementioned techniques, reports satisfactory accuracy on ImageNet classification with only 1-bit weight, 2-bit activation, and 6-bit gradient. The error obtained in EC is floating-point, thus making this method only suitable for inference. At the same time, DoReFa introduces tanh function in quantization, introducing more hardware overhead. The extreme case for quantization is to use binary activation and weight for convolution computations, such as BNN (Hubara et al. 2016) and XNOR-Net (Rastegari et al. 2016). BNN directly quantizes the weight to the sign, while XNOR-Net introduces a scaler per output channel on the weights' sign. They both use floating-point gradient and error for EC/GC/WU, making them a better choice for inference than training. CIM inference chips based on XNOR are reported in Yin et al. (2020a) and Liu et al. (2018).

A promising algorithm for training called WAGE is proposed in Wu et al. (2018), whose name comes from the fact that the quantization is applied to weight, activation, gradient, and error. An advantage of WAGE is that it uses fixed-point quantization between the range of $(-1,1)$ for weights and activations with a precisely pre-calculated factor, which is friendly to hardware implementation. The error is scaled with its max value, which introduces an overhead, but there is no need to scale back and thus is a relaxed version of dynamic fixed-point. With mixed-precision weight and stochastic quantization, WAGE could reduce the precision of weight, activation, gradient, and error to 2, 8, 8, and 8 bits with no loss for moderate tasks such as CIFAR-10 classification. Thus, in this algorithm, the weight used for FF and EC process is 2-bit while an 8-bit high precision version is needed for weight update according to 8-bit weight gradient. In the actual hardware implementation, the work (Sun et al. 2018) uses the "volatile" gate voltage of ferroelectric field-effect transistor (FeFET). The problem with WAGE is that it suffers from performance loss when scaled to more complicated tasks such as ImageNet classification. An improved WAGE version is proposed with dynamic fixed-point used to replace original fixed-point quantization, achieving improved accuracy but increasing hardware implementation complexity. It is an attractive method for training as it considers the quantization for all the parties. Thus, some CIM accelerators for training are proposed based on this algorithm (Jiang et al. 2020b).

While these quantization methods could be applied to the CIM architecture in principle, they are mainly developed for the digital system where all the calculations are done in full precision, even if precision might be low. However, the MAC of CIM is implemented in the analog domain and needs to employ

ADCs to convert the analog partial sum to the digital value. Considering the dynamic range and circuit complexity, it is impractical to use a full-precision ADC according to output precision in most parallel reading cases. Furthermore, since array partitioning is necessary for a large weight matrix, the quantization error (from sub-arrays) introduced by the ADC will be further accumulated, which is usually not considered in the quantization algorithms mentioned above. Thus, when mapping the training/inference algorithm to the CIM system, one needs to consider the ADC quantization effect on the performance. This will be discussed in detail in the latter section when introducing the ADC circuits.

Hardware Implementations for CIM Architecture

Device Technologies

Memory device is served as the core component of CIM. Information could be stored in a memory device through changes of charge value, material state, atom arrangement or orientation. In principle, CIM could be implemented by any device technology. From the perspective of the device structure, CIM designs can be roughly sorted into three categories: SRAM-based, two-terminal eNVM-based, and three-terminal eNVM based. The two-terminal eNVM devices include RRAM (Chen 2020), PCM (Kim and Lee 2020), and spin-torque transfer magnetic random access memory (STT-MRAM) (Ikegawa et al. 2020) and three-terminal eNVM devices include FeFET (Mikolajick et al. 2020). SRAM is considered as a mature candidate from the technology availability's perspective, while eNVM technologies are promising to CIM due to their high density (e.g., with multi-bit per cell) and nonvolatility (especially for the standby-frequent edge devices). There are also some CIM designs based on embedded DRAM (Li et al. 2018b) or 3D NAND flash (Lue et al. 2019). Since embedded DRAM has not been widely used and Flash memory has limited peripheral logic unless using a 3D stacking strategy (Shim and Yu 2021), they are not covered here (the readers could check the references for details if interested).

SRAM

SRAM is widely used as the on-chip buffer in microprocessors and has enjoyed the benefits of scaling together with logic transistors. With aggressive scaling, large on-chip SRAM capacity has been demonstrated (e.g., 256 Mb SRAM demonstrated at 5 nm (Yeap et al. 2019)), which is possible to hold most of the weights on-chip. Besides, because SRAM is more economical to write than eNVMs, it is suitable for training tasks that need to update the weights frequently.

In theory, the conventional 6 T SRAM (shown in Fig. 11a) could be directly used for CIM computation. For the MAC operation, the BL and BLB will be first pre-charged. Then, the input will be applied on transistor T1 and T2 through WL with weight bit represented by the value at node Q. When the input is "1", the cell will attempt to charge BL if Q is "1" and discharge BL if Q is "0". When the input is "0"

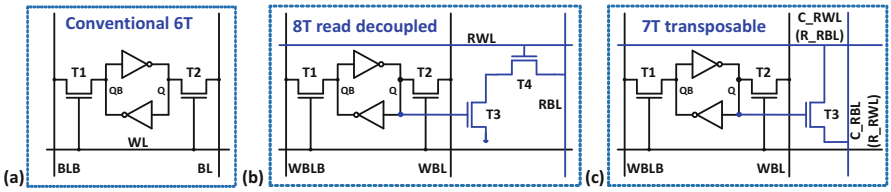


Fig. 11 SRAM cells used in CIM: (**a**) conventional 6 T SRAM; (**b**) read decoupled 8 T SRAM; (**c**) transposable 7 T SRAM

for each row, T1 and T2 will be off, and Q/QB has no contribution to the current on BL and BLB. When multiple rows are turned on, BL will decay from the power supply VDD with different rates, and its voltage when the sense amplifier (SA) is enabled represents the analog MAC result. This structure has two limitations. One is the reliability issue called read-disturb when multiple rows are activated simultaneously. For the regular 6 T SRAM read, only one cell is activated (thus, one discharge path is connected to BL or BLB), causing either BL or BLB to decay. The WL will be closed as long as the voltage difference is big enough for the SA output to flip. Thus neither BL nor BLB will drop to a very low voltage value. However, for the CIM mode, several cells will contribute to the discharge paths on both BL and BLB, making the discharge rates much faster. When either BL or BLB is too low, it will flip the nodes storing "1" that is connected to it. Since the voltage of BL represents the analog MAC result, a high dynamic range is preferred. Read-disturb limits the design range of the 6 T SRAM-based CIM. Another issue for the 6 T cell is the asymmetric data pattern for partial sum accumulation. The product of input "1" (applied to the pass-gate) and weight "0" (stored in the cell) has a different impact on discharge current than the product of input "0" (applied to the pass-gate) and weight "1" (stored in the cell). The analog output representing "0" varies according to different input and weight data patterns. To eliminate the reference mismatch for ADC quantization, input-aware dynamic reference generation is employed (Huang et al. 2020a).

To be compatible with in-memory computation, the innovation of bit-cell design is desired. Thus, a more practical choice for SRAM-based CIM is the 8 T read-decoupled bit cell, as shown in Fig. 11b, at the noticeable expense of additional layout area. The 8 T cell is originally designed for memory used to split the write and read port to eliminate the write and read margin trade-off problem of the 6 T SRAM. The two extra transistors (T3 and T4) in series on the read port form a natural structure to support the multiplication of weight bit at node Q and input applied through RWL. Only when both the input and weight bit are "1" will there be a discharge path connected to RBL. In this case, the data pattern for weight and input is symmetric since both are represented by the gate voltage. Also, since the read is decoupled, it will not affect the data stored in the cell and has a larger dynamic range.

Both the original 6 T and 8 T cells are not efficient for the training since the input and output direction is not transposable. They could be modified to make transposable read possible with extra transistors and BL/WL added. Some memory designs will rotate the direction of the RWL and RBL in the 8 T read decoupled cell, so the column read is supported through the additional read port while the row read is still done by the original 6 T, suffering from read-disturb and asymmetric data pattern problems. A low overhead 7 T transposable SRAM cell with decoupled read on both directions is used for CIM computation in the work (Jiang et al. 2020a) as shown in Fig. 11c. For the FF process, input is applied on C_RWL while weight is applied on the gate of T3. When input and weight are both "1" a charge current will be contributed to R_RWL. On the other hand, when the input is "0", the input is float instead of connected ground to avoid the asymmetric data pattern. C_RWL and C_RBL exchange their roles for the EC process: C_RWL acts as R_RBL, and C_RBL acts as R_RWL. R_RWL is enabled as neuron input and R_RBL is used as bitline for partial sum read-out. Column and row paths can have separate sets of ADCs for easy routing or one shared set of ADCs for area reduction. The drawback of such a 7 T design is that the sneak paths may exist if the unselected rows/columns are left floating. Other bit-cell variants are proposed to fulfill specific operational requirements. For example, to map XNOR-Net to the CIM cell, both input and weight are binarized to $+1/-1$. Although this two-state value could be represented by a single bit, the multiplication between two bits is XNOR instead of AND. A split-6 T cell design was proposed (Khwa et al. 2018) to support XNOR operation in SRAM cells. Similar work also uses this structure to integrate XOR ciphers into SRAM CIM architecture (Huang et al. 2020a). A variant of the design is 8 T-XNOR bit-cell (Liu et al. 2018), where two additional pass-gate transistors crossly-coupled BL and BLB to implement XNOR. Another design that implements XNOR-Net uses a custom 12 T bit-cell with significant overhead (Yin et al. 2020a).

Two Terminal eNVM

Compared to SRAM, eNVM is preferred for power-constrained edge devices as the chip could be controlled by dynamic power gating and perform instant inference operations once receiving awake signals. Despite that eNVM lags behind the technology scaling than SRAM, these nodes are sweet spots from low-cost perspectives and eNVMs are still much higher density than SRAM at the same technology node. As aforementioned, two-terminal eNVMs primarily include RRAM, PCM, and STT-MRAM. RRAM and PCM support multi-bit per cell, while STT-MRAM could only support binary operations. Therefore, most CIM architectures take advantage of RRAM and PCM.

The conventional 1-transistor-1-resistor (1T1R) cell is based on the typical foundry embedded memory design rule where WL is horizontal with both BL and SL vertical (in parallel), as shown in Fig. 12a. This structure is generally used for inference chip design. The input is applied to WL with SL grounded. Thus the current is summed along BL. For the training, to support in-array bidirectional

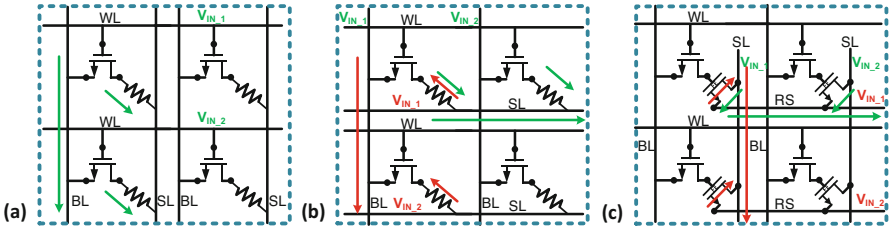


Fig. 12 Structure of eNVM-based arrays: (**a**) 1T1R cells based on the typical foundry embedded memory design rule; (**b**) 1T1R based on a pseudo-crossbar array; (**c**) 1T1FeFET cells based on a pseudo-crossbar array

calculation, a modified 1T1R array that switches BL to horizontal and makes BL and SL perpendicular is referred to as a pseudo-crossbar array (Yu et al. 2015), as shown in Fig. 12b. Input is applied on SL and the current sum along BL for the FF and vice versa for the EC. It is noted that, while for the inference, some works used multi-cell per weight to represent high precision weights, most works for training focused on pure analog cells (Huang et al. 2020b).

Three-Terminal eNVM

Three-terminal NVM devices generally employ the transistor channel conductance to represent the weight, which could be modulated by a tunable threshold voltage. Representative three-terminal NVM is Flash memory based on floating-gate or charge-trap cell. Owing to its extremely high write voltage (>20 V) and long write latency (>10 μs), Flash memory is only applicable for inference design. On the other hand, the emerging ferroelectric field-effect transistor (FeFET) is a more promising solution. By modulating the polarization direction of the ferroelectric layer in the gate stack, the threshold voltage could be changed and thus introduce different channel conductance in the transistor. FeFET shows some highly desired features, such as super high on/off ratio (>1000), relatively short programming pulse width (<50 ns), and smoothness in the weight-update curve. Most importantly, scaled FeFET with the field-driven mechanism will consume much lower programming energy (∼fJ/bit) compared to RRAM or PCM (∼pJ/bit), which is current-driven.

The basic cell structure of FeFET for CIM is shown in Fig. 12c. During CIM read operation, the input is applied to SL while the current is summed along BL. This is similar to the configuration of the transposable CIM operation and thus could be directly used for training. Unlike the 1T1R cell, which has a transistor in serial with the resistor representing weights, the FeFET has an extra transistor connected to its gate. The additional transistor is used as the selection control to skip the unselected rows if the input is zero, as the weight being "1" in FeFET means negative threshold voltage thus a floating gate voltage offered by the extra transistor is necessary to turn off the channel.

Overcoming the Non-idealities from eNVM

The performance of the CIM-based designs could be highly affected by the properties of the device. Compared to SRAM, eNVM suffers from more device non-idealities, including limited on/off ratio, nonlinear and asymmetric conductance tuning behavior, conductance variation, device-to-device variation (D2D), cycle-to-cycle variation (C2C), and write endurance. First of all, the eNVM cells have limited dynamic ranges, which defines the on/off ratio between the maximum conductance (G_{max}) and the minimum conductance (G_{min}). As the cells use different conductance to represent different weights, the dynamic range will limit the number of levels the cell could represent. For example, STT-MRAM usually has a small on/off ratio and can only be used as binary cells, while ReRAM could be used as multi-level cells. Another issue caused by the small dynamic range is the difference between the currents contributed by G_{max} and G_{min} is not big enough. Normally, the weight "0" is represented by G_{min} and 0 V represents input "0". In this case, input "0" will contribute no current while the G_{min} with non-zero input will still cause some current. If the on/off ratio is big, the current contributed by non-zero input and weight "0" could be neglected. Otherwise, the G_{min} will cause the input-dependent analog output shift problem. To solve this problem, a dynamic reference generation scheme (Chen et al. 2018a) or a reference-column-based compensating scheme (Luo et al. 2020) could be employed. While one dynamic range is defined as the average case for one type of device, different cells of the same type could have conductance variation. Luckily, this variation is tolerable from two aspects. On one side, the neural network itself is noise-robust to some extent. On the other side, this variation could be mitigated by the write-verify scheme to write the cell (Yin et al. 2020b).

The impact of these non-idealities on performance also varies by different applications. For applications where the data only needs to be transferred into memory arrays without frequent updates like DNN inference, accurate conductance programming is crucial, thus write-verify scheme is usually used to enhance the quality of programming. Besides, the retention problem of eNVMs, i.e., the resistance drift, will introduce errors of weight values, resulting in inference accuracy degradation. To address the challenge caused by resistance drift, one potential approach is to reduce the drift coefficient through device engineering (Giannopoulos et al. 2018). Another solution is to compensate the drift effect by generating a time-dependent correction term to cancel the deviation caused by the reduction of conductance while introducing significant overhead for additional storage and computations (Ambrogio et al. 2019).

On the other hand, for other applications that involve frequent weight updates like DNN training, eNVM-based CIM suffers more from nonlinear/asymmetric programming, D2D variation, and C2C variations. The eNVM cells are usually viewed as analog synapses instead of multilevel cells when it comes to training. The number of levels defines the smallest step that could be taken to update the cell value. Thus, the dynamic range still matters, and the higher, the better. On the contrary, retention is no longer a problem since the cell is updated frequently.

One critical problem for training with eNVM devices is the asymmetric and nonlinear weight update. For most eNVMs, the trajectory for potentiation and degradation is nonlinear and asymmetrical (Chen et al. 2018b). As reported by Sun and Yu (2019), the nonlinear but symmetrical update of weight conductance will not cause a big accuracy loss for training. At the same time, the nonlinearity combined with asymmetry will degrade the training performance a lot. An adaptive momentum (Huang et al. 2020b) method could be used to compensate the non-ideal conductance tuning. D2D variation, caused by varying device nonlinearity, could also be compensated with a similar scheme. Circuit techniques such as adding auxiliary transistors in the bit-cell (e.g., 2 T-1FeFET (Sun et al. 2018)) can also make the update more linear. C2C variation is more problematic, thus further device engineering to suppress the variation is desired. Besides, high device endurance is preferred to support training on chip. It should be emphasized that the requirement on the endurance cycle is also task-dependent, i.e., the number of weight update iterations needed for convergence.

Circuit Techniques for CIM

According to the basic structure of CIM sub-array (as shown in Fig. 1c), by applying voltages V_i simultaneously to crossbar rows, multiplication is performed using Ohm's law between the device conductance and V_i to produce current I_i. Then I_i from multiple rows is summed up through the column, representing the dot-product. The possible bidirectional array design for training has a similar computing manner, and the only difference is that the summing directions will be perpendicular. CIM arrays typically have three design components: (1) memory array; (2) input encoding; (3) output sensing. Circuit techniques for those components will be introduced respectively.

Memory Modification

In old technology nodes (e.g., 65 nm), there is room to modify the bit cell and reroute interconnection. But in advanced technology nodes (e.g., 28 nm or beyond), foundries typically do not offer exceptions for making these changes for memory design. Although it is possible to implement SRAM using the logic design rule, it will not be as compact as the compact memory array design. Therefore, a viable approach is to group the 6 T SRAM rows into memory banks and embed a row of compute cells between banks to enable more sophisticated functions, as shown in Fig. 13a. In this configuration, the 6 T SRAM is used as a memory cell only, and the real computation is done in the compute cells. A group of compact 6 T SRAMs is connected through local-bitline/local-bitline-bar (LBL/LBLB) for each bank. The data from the 6 T SRAM cell is also sent to the compute cells through LBL/LBLB. Thus, compute cells for banks could work in parallel during compute mode without sacrificing throughput. All the banks are connected through global-bitline/global-bitline-bar (GBL/GBLB). The normal cell read and write for the

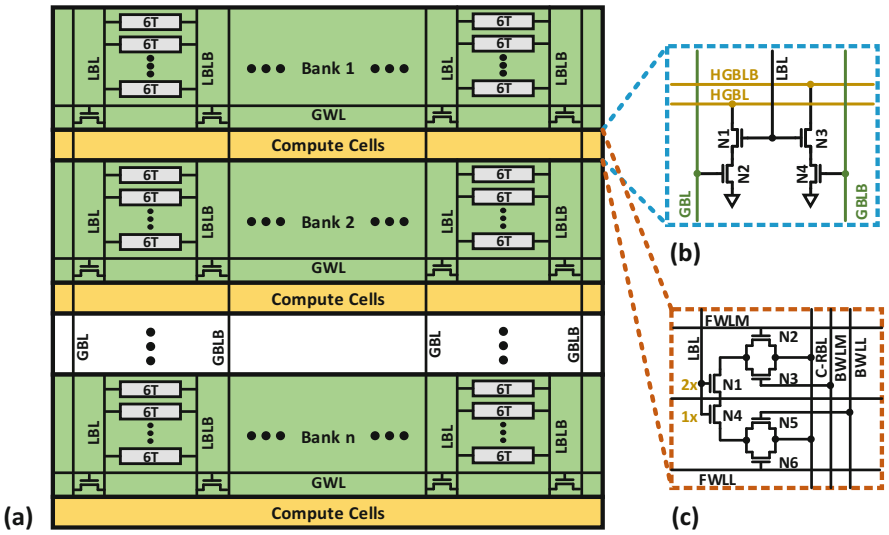


Fig. 13 Memory modification for embedded computing cells to foundry rule defined 6 T SRAM arrays: (**a**) memory banks with computing cells inserted; (**b**) compute cells supporting 4-bit input; (**c**) compute cells supporting transpose read

memory array could be done through the GBL/GBLB. Figure 13b,c shows two typical local-computing cell design (Si et al. 2020; Su et al. 2020). The design in Fig. 13c is taken as an example to explain the functionality. In each input cycle, a 2-bit input is applied to the gates of *N1* and *N4*, corresponding MSB and LSB, respectively. The *N1-N3* pair and *N4-N6* pair correspond to MSB multiplication and LSB multiplication of input, respectively. The transistor width of *N1-N3* is designed to be twice of *N4-N6*, which enables the *N1-N3* pair to produce a discharge current twice of the *N4-N6* branch. When the two currents of both *N1-N3* and *N4-N6* pairs are summed up, the voltage swing contributed by the computing cell is proportional to the result of multiplying 2-bit input and 1-bit weight. For the backward calculation, the computation is performed in a perpendicular direction. The advantage of such memory modification is that multi-bit MAC operation per cycle or additional functionality (e.g., bidirectional read) could be implemented due to the flexibility of computing cell design, enabling the use of foundry-provided compact-rule cells.

Input Encoding

The inputs of neural networks are usually mapped as voltage pulses in the CIM system. The input encoding scheme can be sorted into three types: amplitude-based modulation, time-domain modulation, and binary encoding (as shown in Fig. 14). For amplitude encoding, digital-to-analog converters (DACs) are essential to encode the digital inputs as a voltage pulse of different amplitudes. Typical DAC

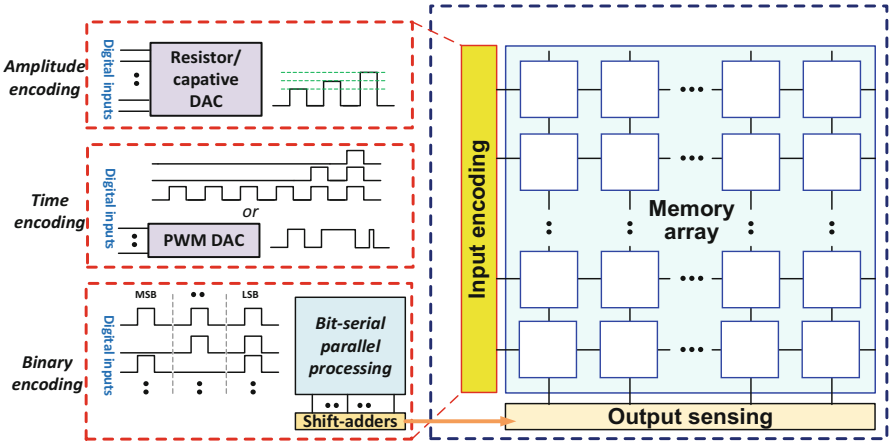


Fig. 14 Input encoding schemes for CIM

designs, including capacitive circuits and resistive ladders, will introduce additional area overhead and power consumption. Multi-level input voltage could also be implemented with multiple power supply sources to perform the multi-bit operation. The introduced area overhead is small compared to the DAC approach, increasing the load of the on-chip power unit. In general, the downside of such amplitude-dependent encoding is that the number of voltage levels is limited by the narrow signal swing in the voltage domain and the current-voltage linearity of the memory cell.

For the time-domain modulation, the number of pulses can be directly encoded as the input information. Alternatively, a pulse-width-modulation (PWM)-based DAC can be hired to encode multi-bit inputs as varying pulse-widths. Naturally, the computation time of time-dependent encoding is much longer than amplitude encoding. However, compared to amplitude encoding, time encoding is much less affected by the current-voltage nonlinearity, which makes it appealing to achieve more accurate results. Additional advantages of PWM encoding are its simple single voltage bias design and the ease of signal propagation between sub-arrays without explicit ADC at the output node (Chang et al. 2019).

For binary encoding, multi-bit inputs are sent sequentially to the CIM array (i.e., bit-by-bit) and processed in a bit-serial parallel fashion. Additional circuits, such as registers and shifters, are required to combine the sequential data. In such a way, input DACs could be totally eliminated while hiring less expensive digital circuits. Compared to time-dependent encoding, binary encoding offers higher throughput, which is widely used in CIM designs. In practice, since the dynamic range of the outputs from the circuits/devices is limited, inputs are usually limited to $1 \sim 2$ bit per cycle, no matter which input encoding method is used.

Output Sensing

For the output sensing in most of today's CIM designs, the memory array is usually equipped with ADCs to convert analog MAC values to digital outputs. The digital signal can be passed to the peripheral circuitry for further processing steps such as activation function/pooling and then sent to the next array as the input. This mixed-signal computing scheme offers scalability toward tiled hierarchical designs via interconnect buses or network-on-chip while introducing additional power dissipation and area overhead in the necessary data conversion at the array outputs.

The raw output of a MAC operation performed by the CIM crossbar is typically in the form of a current. Current-mode ADCs are required for direct sensing, while voltage-mode ADCs can be employed after the current-to-voltage conversion. A resistor divider or a trans-impedance amplifier (TIA) is commonly used to convert the current to voltage. An ADC design plays an important role in CIM, as it has a substantial contribution to the area/energy consumption of the CIM array. From the algorithm's viewpoint, although binary neural networks such as XNOR-Net (Rastegari et al. 2016) may greatly reduce the required memory capacity and even eliminate ADCs with simple binary sense amplifier (SA), multi-bit precision is a more generic setting for large-scale DNNs to avoid inference accuracy degradation. To reduce the overhead of ADCs, several alternatives have been proposed. One is to share the ADC with multiple columns. As a penalty, the parallelism of CIM will be reduced with lower throughput. The other solution is to lower the ADC precision. However, the quantization loss of partial sum may hamper the accuracy performance. Therefore, the choice of ADC topologies and configurations is critical to the design of the CIM architecture. Several candidates of ADCs, such as Flash-ADC (Yin et al. 2020a) and successive-approximation-register (SAR)-ADC (Chen et al. 2018a), have been explored in prior CIM designs due to their simplicity and suitability for low to medium precision.

As shown in Fig. 15a, Flash-ADC comprises cascading comparators. For an N-bit converter, the circuit employs $2^N - 1$ comparators. The thermometer code generated by comparators is then encoded into the digital output code. Flash-ADC is the fastest ADC design in principle but consumes exponentially larger power and area when the precision increases. Thus, Flash-ADC design in the resource-constrained CIM architecture is usually restricted to low precision, such as 3-bit or below (Liu et al. 2018), to ensure good performance and is mostly used for small-scale arrays or partially turned-on rows.

While Flash-ADC offers high speed, SAR-ADC is competitive due to its linearly increased area/energy cost when the precision becomes higher. As shown in Fig. 15b, SAR-ADC hires a single comparator but performs a one-bit comparison only in one internal clock. Based on the binary search, the SAR logic (implemented with multi-stage shift registers) will adjust the references dynamically and makes the comparison in a bit-by-bit fashion. This sequence continues all the way from MSB to LSB. Once all bits are done, the conversion is complete, and the N-bit digital

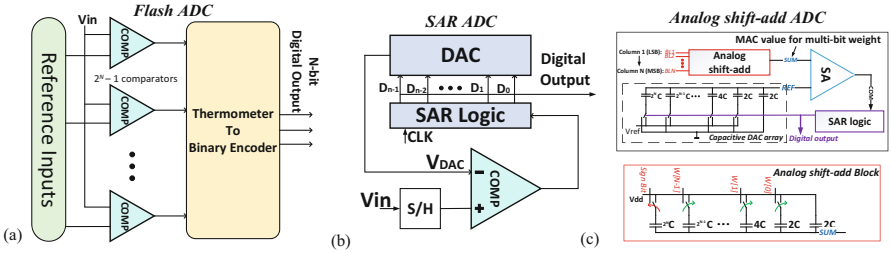


Fig. 15 (**a**) Diagram of Flash-ADC; (**b**) diagram of SAR-ADC; (**c**) diagram of analog shift-add ADC. (With permission of Jiang et al. 2021)

output is available in the register. Typically, a capacitive DAC array that exploits the charge redistribution is used to generate the analog reference voltage. To support successive current-mode sensing, the multi-level current sense amplifier (ML-CSA) is proposed in Chen et al. (2018a). The reference current (generated by the reference array) input of each sensing step is selected by the previous output signal.

To reduce the ADC overhead, an analog shift-add approach for binary input encoding is demonstrated in Su et al. (2020). The top-level diagram of the analog shift-add ADC is shown in Fig. 15c. Here the shift-add process for weight precision is moved prior to the ADC and is being conducted in the analog domain, which is performed in a capacitor array. The charge redistribution nature of the capacitor array is exploited to perform the weighted accumulation before ADC quantization. Once the charge redistribution is stabilized, the pre-shifted and added MAC value would be quantized by regular SAR-ADC to generate the final output that already contains the weight significance. This design effectively eliminates the digital shift-adders for multi-bit weight computation and reduces the MUXs of multiple columns, thus improving the throughput and energy efficiency under the same area constraint.

As aforementioned, quantization loss introduced by ADCs may decrease the accuracy performance. To determine the minimally required ADC precision, the impact of ADC quantization loss should be evaluated. In general, the smaller the sub-array size and/or the lower the cell precision, the more relaxed the requirement is on the ADC quantization. Figure 16 shows the analysis of the ADC quantization effect on a CIFAR-10 classification task with a VGG-8 network (Peng et al. 2019). For the smallest partial sum range case, which is a 64 × 64 array and 1-bit per cell, the full precision of partial sum is 6-bit thus 5-bit ADC will introduce 1-bit loss. It could be seen that for the same sub-array size, when the bit per cell increases, the accuracy drops if using the same ADC precision. For the same cell and ADC precision, the accuracy will drop accordingly as the sub-array size increases. In a specific case, 5-bit ADC is necessary for 256 × 256 array size with a 4-bit/cell design.

Figure 17 shows the accuracy performance vs. ADC precision for the conventional ADC designs (representing both Flash- and SAR-ADC) and the proposed

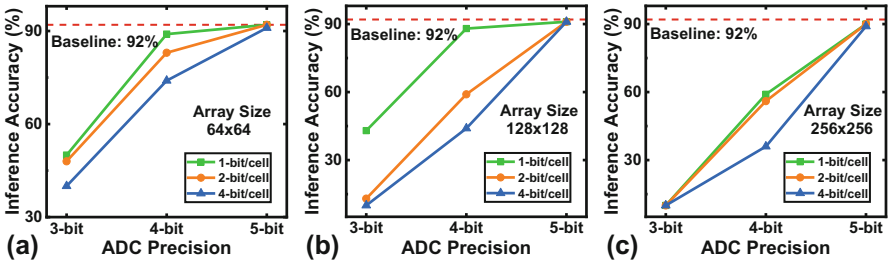


Fig. 16 Accuracy performance vs. ADC precision for different memory array configurations. Simulations on VGG-8 network for CIFAR-10 dataset. (With permission of Peng et al. 2019)

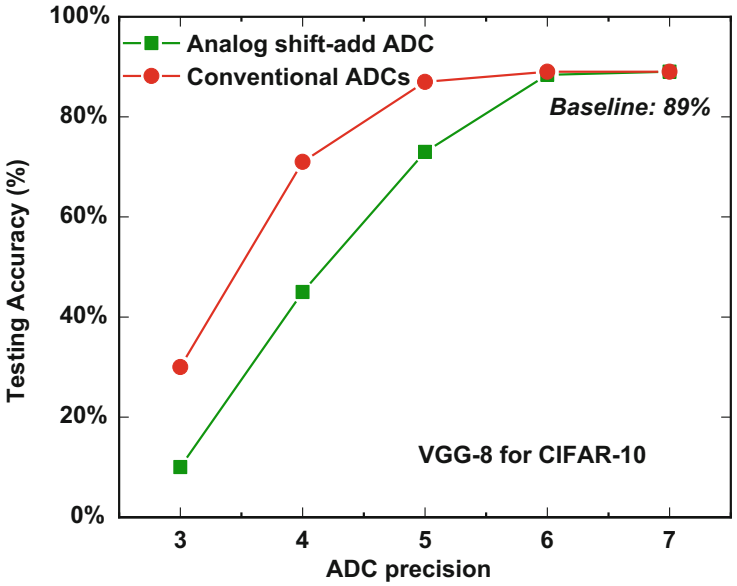


Fig. 17 Accuracy performance vs. ADC precision for analog shift-add ADC and conventional ADCs. Simulations on VGG-8 network for CIFAR-10 dataset

analog shift-add ADC (Jiang et al. 2021). The assumption is 4-bit weight precision (one bit per cell) with 512 × 512 memory sub-array, which means the full precision of each column's partial sum is 9-bit. Here Flash-ADC and SAR-ADC are grouped together since they have the same dataflow in which ADCs first quantize partial sums, and then digital shift-add modules are employed to accumulate digitized partial sums. Oppositely, the analog shift-add ADCs first weigh and sum up the analog multi-bit MAC outputs and then quantize the final output that already contains the weight significance. These two different quantization approaches are embedded in software simulation to investigate the impact of ADC quantization loss on inference accuracy performance for the CIFAR-10 dataset.

For conventional ADCs, 3-bit quantization loss is tolerable. For analog shift-add ADC, the full input precision for the partial sum is 13-bit (9-bit from column partial-sum and 4-bit from shift-add for weight precision). From the plot, it is observed that 6-bit analog shift-add ADC could maintain baseline accuracy, actually tolerating 7-bit quantization loss. Analog shift-add ADC could tolerate more quantization loss because quantization loss only happens at the final stage for analog shift-add ADC. On the contrary, conventional ADCs have quantization loss on each partial sum and then accumulate these quantized partial sums by shift-add, which means errors will be accumulated. In sum, the analog shift-add before the ADC could preserve more information, while the digital shift-add after the ADC may lose some residual information, resulting in more errors in the final output. The trend would be similar for different networks and tasks. Nevertheless, the ADC-induced accuracy loss requires empirical simulations and case-dependent analysis for different datasets. To reduce the ADC overhead and associated design efforts, some designs only activate a part of the sub-array in parallel per cycle and use multiple cycles to finish the MAC for the full array (e.g., (Li et al. 2021)). The ADCs used in these designs could introduce no quantization loss theoretically since the full precision of the par ADC quantization raises new challenges beyond quantization loss. Unlike the precise quantization on the digital partial sum, the ADC quantization on the analog signal could be noisy due to both static and dynamic uncertainties. The static uncertainty could be introduced by the process variation, dominated by the ADC offset caused by transistor mismatch. The ADC offset may cause quantization errors when converting the analog partial sum to the digital signal. Thus it may noticeably degrade the inference accuracy and cause different chip instances to have different inference results even for the same input. As shown in Fig. 18, a 5-bit Flash-ADC and a 5-bit SAR-ADC are evaluated by SPICE Monte Carlo simulations using a foundry's 40 nm PDK assuming local variations for an RRAM CIM array. The error caused by the ADC offset will vary for different ADC types, sense amplifier size (W/L), and the level of ADC output. Flash-ADC introduces less error than SAR-ADC since with the same size sense amplifier on the same level since Flash-ADC has multiple SAs to sense different levels in parallel. Thus the error caused by them may compensate for each other, considering the summation as the thermometer-to-binary (TM2B) encoder. For the same type of ADC, smaller size for SA will be affected more by the process variation, thus introducing more

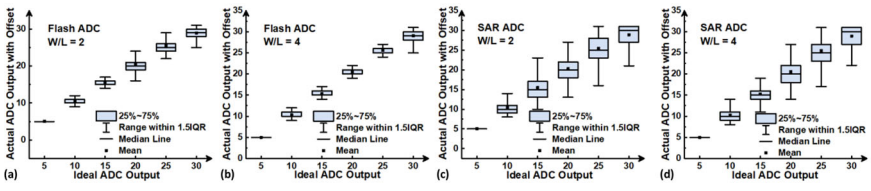


Fig. 18 Simulated ADC output with offset based on the sense pass rate for different W/L ratio. Simulations on VGG-8 networks for CIFAR-10 dataset. (With permission of Huang et al. 2021)

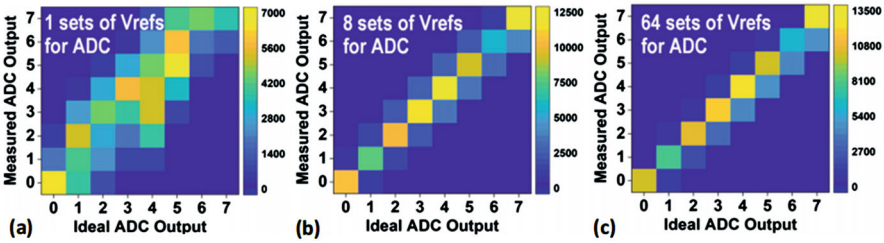


Fig. 19 Measured ADC vs. ideal ADC outputs from a 90 nm RRAM CIM macro showing the effectiveness of the reference fine-tuning. (With permission of Yin et al. 2020b)

error. The quantization error increases with the increase of the partial sum because when the column current goes up, the effective resistance of the RRAM pull-down networks decreases. Therefore, the voltage levels to be sensed are closer to each other. Since the accuracy of SA highly decides the ADC error, one way to solve this problem is to use more advanced offset-canceled SAs that could decrease the mismatch caused by variation (Xue et al. 2019). On the other hand, since the static offset will not change over time once the chip is fabricated, it could be compensated by fine-tuning the ADC references (Yin et al. 2020b) or the model weights with hybrid retraining on-chip (Huang et al. 2021). Figure 19 shows the error pattern of eight 3-bit ADCs shared by 64 columns of an RRAM-based CIM chip fabricated at 90 nm (Yin et al. 2020b). Figure 19a shows that when all the ADC use the same reference set, the measured ADC output is much dispersed. The error decreases significantly when each ADC has its own reference set, as shown in Fig. 19b. Further increasing the number of reference sets to one set per column will introduce marginal improvement, as shown in Fig. 19c, since one reference set per ADC could fully compensate for the SA offset. However, it will introduce substantial hardware overhead if every ADC reference needs fine-tuning. Another method to compensate for the ADC offset is to perform the model retraining to adapt the weight patterns to each chip's ADC offset pattern. The hybrid retraining method will perform FF on-chip while executing the EC and GC off-chip by software. Thus, the ADC offset will be incorporated during the weight fine-tuning. Figure 20a, b shows an example of CIFAR-10 classification on the VGG-8 network with 2-bit weight and 8-bit activation. The software accuracy after quantization is ~92%. Considering the ADC offset, the accuracy drops to 89.48% for W/L = 2, 90.26% for W/L = 3 and 89.91% for W/L = 4 when Flash ADC is used, while the accuracy drops to 81.65% for W/L = 2, 84.16% for W/L = 3 and 86.14% for W/L = 4 when SAR-ADC is used. After the model retraining with weight fine-tuning, the accuracy will recover to ~91% for all the cases. Dynamic noise is mainly introduced by the temporal effect, such as device read noise, VDD disturbance, etc. The dynamic noise is much more difficult to be eliminated as it varies over time. In the ADC quantization case, it means the current/voltage margin between levels in the sensing node needs to be large enough to tolerate the noise. Figure 21 shows an example

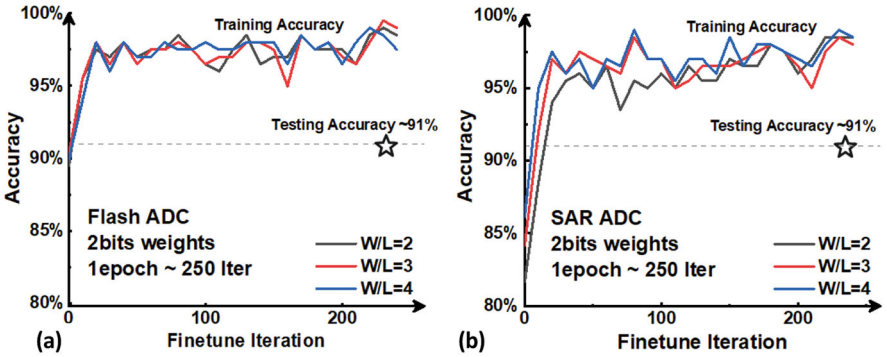


Fig. 20 Retraining accuracy curves for inference chips equipped with Flash-ADC and SAR-ADC. (With permission of Huang et al. 2021)

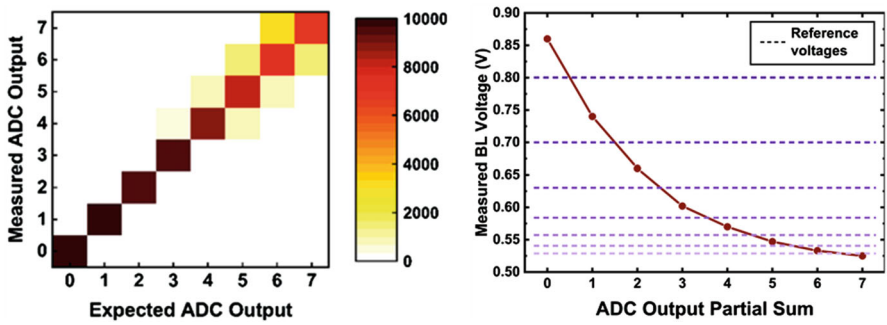


Fig. 21 Measured ADC vs. ideal ADC outputs with dynamic noise only from a 40 nm RRAM CIM macro and the nonlinear mapping between the BL voltage and the expected ADC output partial sum. (With permission of Li et al. 2021)

of 3-bit ADCs input/output mapping table for a 40 nm RRAM CIM array with references fine-tuned measured over 10,000 input test vectors, which means ADC offsets are already compensated (Li et al. 2021). In an ideal condition, the measured ADC output should be the same as the measured one. However, dynamic errors are introduced because of the dynamic noise. The noise affects higher partial sum levels more than lower ones. This is because as the partial sum gets higher, the margin between levels shrinks due to the nonlinear relationship between the sensing node voltage and the partial sum current in a resistive divider configuration (Fig. 21b). To improve the linearity, a current source with a feedback amplifier is proposed to linearize the sensing voltage steps between the expected partial sum (Yoon et al. 2021). In addition, the noise-aware training (e.g., with noise injection) could be used to converge the DNN model in a local minimum that is insensitive to the hyperparameter deviation when the model is deployed to the hardware (Long et al. 2019). From both software accuracy and hardware cost point of view, it could be seen that the performance of ADC is the key bottleneck for the CIM architectures.

Frameworks for Evaluating CIM Designs

Many factors, from device to algorithm, have to be considered while designing a CIM architecture. Therefore, to facilitate architectural exploration and system integration of CIM-compatible technologies and design approaches, a comprehensive and validated framework/tool/simulator is essential to evaluate both the hardware and algorithm performance. CACTI (Wilton Steven and Jouppi 1996) and NVSim (Dong et al. 2012) are the first developed memory-oriented simulators designed for SRAM/DRAM and nonvolatile memories, respectively. Compared to those early-stage simulators which only support memory applications, MNSIM (Xia et al. 2016) proposed the first behavior-level simulation framework for neuromorphic computing. NeuroSim (Chen et al. 2018b), which was inspired by the principles of CACTI and NVSim, is an open-source simulator for benchmarking in-memory computing or near-memory computing architectures in terms of hardware performance metrics such as area, latency, dynamic energy, and leakage power. Various device technologies are supported, including SRAM and eNVMs (e.g., RRAM, PCM, STT-MRAM, FeFET). The hierarchy framework of NeuroSim is organized from the device level to the circuit level (memory array, periphery circuit modules such as ADCs), to the tile level (built up by multiple sub-arrays, local buffer, and interconnect routing) to chip level (built up by tiles, global interconnect, and buffer). Thanks to the modular programming of the simulator, NeuroSim is friendly for customizing and exploration with any new device technologies or circuit techniques.

While hardware performance is the most important metric for the CIM-based architecture design, software performance must be maintained as a precondition. Thus, simulators to evaluate the software performance under the CIM hardware non-idealities are another important category of tool development. Different simulators have been proposed with different deep learning machine learning frameworks. Based on the preference of the developer, different non-ideal effects might be covered with different modeling methods. For example, DL-RSIM (Lin et al. 2018) is proposed as a TensorFlow-based simulator to evaluate the software performance of ReRAM-based Accelerators. It is aimed at helping chip designers to pick more reliable options from the big design space of the CIM architecture. Besides the eNVM's non-idealities such as on/off ratio and conductance variation, DL-SIM is highlighted by introducing the non-ideal effects of sense amplifier into the simulation. As mentioned before, the ADC plays a very important role in the CIM operation. However, most simulators focus on the non-idealities of the devices instead of the peripheral circuits. Thus, this simulator gives a quite comprehensive evaluation on the circuit level. On the other hand, a PyTorch-based simulator for accuracy evaluation called PytorX is also introduced (He et al. 2019). The non-ideal effects it covers include IR-drop (i.e., wire resistance), Stuck-At-Fault (SAF) defects, thermal noise, shot noise, and Random Telegraph Noise (RTN). Based on the characteristics of these non-idealities, they are modeled as deterministic or stochastic noise in the inference of the neural network. Besides mapping and evaluating crossbar-based neural networks, this simulator introduces end-to-end

training with noise injected to recover the performance loss caused by the non-ideal effect of the circuit. Thus, this simulator could help designers to expand the deterministic or stochastic noise design space for CIM design. With Pytorch and TensorFlow wrapper, DNN+NeuroSim framework (Peng et al. 2019) can also emulate DNN inference/training accuracy considering hardware limitations and device non-idealities. Furthermore, the newly released 3D+ NeuroSim can support electrical-thermal co-simulation of 3D-integrated CIM accelerators.

In general, the shift from conventional von Neumann architectures to CIM architectures necessitates the rethinking of the whole framework from hardware to software. The simulation platforms should bridge the gap between different abstraction levels, including device, circuit, architecture, system, and accuracy simulation. Frequent update/calibration and integrating more functionalities are desired to validate CIM implementations. With the development of CIM accelerators, the simulators will also become more comprehensive and accurate.

Conclusion

An overview of CIM architectures is provided, especially for deep learning-based image classification workloads. So far, considerable advances have been made (from the architecture level down to the device level) in CIM designs. At the architecture level, network mapping and pipeline design have been comprehensively explored to optimize the dataflow for DNN computations, promising tremendous throughput and energy efficiency benefits. Besides, possible quantization methods and algorithms are also evaluated on CIM architectures for their effectiveness. At the circuit level, since expensive peripheries such as DACs and ADCs are required for high-level communications, researchers make endless efforts to develop advanced circuit techniques to reduce the area and energy consumption of these modules. At the device level, many emerging device technologies are examined for CIM applications, and SRAM-/eNVM-CIM architectures show their advantages and trade-offs in different aspects including scalability, nonvolatility, multi-bit capability, etc.

However, design challenges remain at each level. At the architecture level, the architecture and dataflow of CIM are still immature and need to be standardized with possible compiler and instruction set architecture (ISA) support. For training complex datasets such as ImageNet, the system performance of CIM architectures is still bounded by off-chip data transfer since massive intermedia data (>GB) is difficult to be fully processed on-chip (Jiang et al. 2020b). Besides, design automation tools for CIM architectures are critical to the performance evaluation considering hardware non-idealities. Benchmark framework such as DNN+NeuroSim (Peng et al. 2019) provides an open-source platform for early-stage design space exploration. At the algorithm level, the CIM-aware deep learning algorithms and hardware-friendly quantization are still desired to support/adapt CIM unique features such as ADC quantization/offset. At the circuit level, more economical peripheral circuits are

preferred, and the non-ideal noises should be further reduced. At the device level, SRAM suffers from leakage power for standby-frequent applications while eNVMs exhibit the weaknesses of asymmetry/nonlinearity in weight update, variations, and expensive write, which are undesired for in-situ training. As device process technology becomes more mature with extensive industrial research and development, further enhancements in device characteristics are expected. Better linearity, more levels of states, fewer variations, and longer endurance are all desired for delivering the promises offered by CIM architectures.

References

Ambrogio S, Gallot M, Spoon K, Tsai H, Mackin C, Wesson M, Kariyappa S, Narayanan P, Liu CC, Kumar A et al (2019) Reducing the impact of phase-change memory conductance drift on the inference of large-scale hardware neural networks. In: IEEE international electron devices meeting (IEDM)

Chang HY, Narayanan P, Lewis SC, Farinha NC, Hosokawa K, Mackin C, Tsai H, Ambrogio S, Chen A, Burr GW (2019) AI hardware acceleration with analog memory: microarchitectures for low energy at high speed. IBM J Res Dev 63(6):8–1

Chen Y (2020) ReRAM: history, status, and future. IEEE Trans Electron Devices 67(4):1420–1433

Chen YH, Krishna T, Emer JS, Sze V (2016) Eyeriss: a spatial architecture for energy-efficient dataflow for convolutional neural networks. IEEE J Solid State Circuits 52(1):127–138

Chen WH, Li KX, Lin WY, Hsu KH, Li PY, Yang CH, Xue CX, Yang EY, Chen YK, Chang YS, Hsu TH (2018a) A 65nm 1Mb nonvolatile computing-in-memory ReRAM macro with sub-16ns multiply-and-accumulate for binary DNN AI edge processors. In: IEEE international solid-state circuits conference (ISSCC)

Chen PY, Peng X, Yu S (2018b) NeuroSim: a circuit-level macro model for benchmarking neuro-inspired architectures in online learning. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 37(12):3067–3080

Chi P, Li S, Xu C, Zhang T, Zhao J, Liu Y, Wang Y, Xie Y (2016) Prime: a novel processing-in-memory architecture for neural network computation in reram-based main memory. In: 43rd annual international symposium on computer architecture (ISCA), vol 44, p 27

Courbariaux M, Bengio Y, David JP (2015) Training deep neural networks with low precision multiplications. In: Workshop contribution at international conference on learning representations (ICLR)

DeHon A (2000) The density advantage of configurable computing. Computer 33(4):41–49

Dong X, Xu C, Xie Y, Jouppi NP (2012) Nvsim: a circuit-level performance, energy, and area model for emerging nonvolatile memory. IEEE Trans Comput-Aided Des Integr Circuits Syst 31(7):994–1007

Elliott DG, Stumm M, Snelgrove WM, Cojocaru C, Mckenzie R (1999) Computational RAM: implementing processors in memory. IEEE Des Test Comput 16(1):32–41

Feng Y, Chen B, Liu J, Sun Z, Hu H, Zhang J, Zhan X, Chen J (2021) Design-technology co-optimizations (DTCO) for general-purpose computing in-memory based on 55nm NOR flash technology. In: IEEE international electron devices meeting (IEDM)

Gao L, Chen PY, Liu R, Yu S (2016) Physical unclonable function exploiting sneak paths in resistive cross-point array. IEEE Transactions on Electron Devices 63(8):3109–3115

Giannopoulos I, Sebastian A, Le Gallo M, Jonnalagadda V, Sousa M, Boon M (2018) 8-bit precision in-memory multiplication with projected phasechange memory. In: IEEE international electron devices meeting (IEDM)

Gokmen T, Onen M, Haensch W (2017) Training deep convolutional neural networks with resistive cross-point devices. Front Neurosci 11:538

He Z, Lin J, Ewetz R, Yuan JS, Fan D (2019) Noise injection adaption: end-to-end ReRAM crossbar non-ideal effect adaption for neural network mapping. In: ACM/IEEE design automation conference (DAC)

Huang S, Jiang H, Peng X, Li W, Yu S (2020a) XOR-CIM: compute-in-memory SRAM architecture with embedded XOR encryption. In: IEEE/ACM international conference on computer-aided design (ICCAD)

Huang S, Sun X, Peng X, Jiang H, Yu S (2020b) Overcoming challenges for achieving high in-situ training accuracy with emerging memories. In: Design, Automation & Test in Europe Conference & Exhibition (DATE)

Huang S, Peng X, Jiang H, Luo Y, Yu S (2021) Exploiting process variations to protect machine learning inference engine from chip cloning. In: IEEE international symposium on circuits and systems (ISCAS)

Hubara I, Courbariaux M, Soudry D, El-Yaniv R, Bengio Y (2016) Binarized neural networks. In: Conference on neural information processing systems (NIPS)

Ikegawa S, Mancoff FB, Janesky J, Aggarwal S (2020) Magnetoresistive random access memory: present and future. IEEE Transactions on Electron Devices 67(4):1407–1419

Imani M, Gupta S, Kim Y, Rosing T (2019) FloatPIM: in-memory acceleration of deep neural network training with high precision. In: ACM/IEEE 46th annual international symposium on computer architecture (ISCA)

Jiang H, Peng X, Huang S, Yu S (2020a) CIMAT: a compute-in-memory architecture for on-chip training based on transpose SRAM arrays. IEEE Trans Comput 69(7):944–954

Jiang H, Huang S, Peng X, Su JW, Chou Y-C, Huang WH, Liu TW, Liu R, Chang MF, Yu S (2020b) A two-way SRAM array based accelerator for deep neural network on-chip training. In: ACM/IEEE design automation conference (DAC)

Jiang H, Li W, Huang S, Cosemans S, Catthoor F, Yu S (2021) Analog-to-digital converter design exploration for compute-in-memory accelerators. IEEE Des Test 39(2):48–55

Jouppi NP et al (2017) In-datacenter performance analysis of a tensor processing unit. In: ACM/IEEE international symposium on computer architecture (ISCA)

Kawahara A, Azuma R, Ikeda Y, Kawai K, Katoh Y, Tanabe K, Nakamura T, Sumimoto Y, Yamada N, Nakai N, Sakamoto S, Hayakawa Y, Tsuji K, Yoneda S, Himeno A, Origasa K, Shimakawa K, Takagi T, Mikawa T, Aono K (2012) An 8Mb multi-layered cross-point ReRAM macro with 443MB/s write throughput. In: IEEE international solid-state circuits conference (ISSCC)

Keckler S, Kunle O, Hofstee P (2009) Multicore processors and systems. Springer, US

Khwa W-S, Chen JJ, Li JF, Si X, Yang EY, Sun X, Liu R, Chen PY, Li Q, Yu S, Chang MF (2018) A 65nm 4Kb algorithm-dependent computing-in-memory SRAM unit-macro with 2.3ns and 55.8TOPS/W fully parallel product-sum operation for binary DNN edge processors. In: IEEE international solid-state circuits conference (ISSCC)

Kim T, Lee S (2020) Evolution of phase-change memory for the storage-class memory and beyond. IEEE Trans Electron Devices 67(4):1394–1406

Kim D, She X, Rahman NM, Chekuri VCK, Mukhopadhyay S (2020) Processing-in-memory-based on-chip learning with spike-time-dependent plasticity in 65-nm cmos. IEEE Solid-State Circuits Letters 3:278–281

Li C, Hu M, Li Y, Jiang H, Ge N, Montgomery E, Zhang J, Song W, Dávila N, Graves CE, Li Z (2018a) Analogue signal and image processing with large memristor crossbars. Nat Electron 1(1):52–59

Li Y, Kim S, Sun X, Solomon P, Gokmen T, Tsai H, Koswatta S, Ren Z, Mo R, Yeh CC, Haensch W, Leobandung E (2018b) Capacitor-based cross-point array for analog neural network with record symmetry and linearity. In: IEEE symposium on VLSI technology

Li W, Huang S, Sun X, Jiang H, Yu S (2021) Secure-RRAM: a 40nm 16kb compute-in-memory macro with reconfigurability, sparsity control, and embedded security. In: IEEE custom integrated circuits conference (CICC)

Lin MY, Cheng HY, Lin WT, Yang TH, Tseng IC, Yang CL, Hu HW, Chang HS, Li HP, Chang MF (2018) DL-RSIM: a simulation framework to enable reliable ReRAM-based accelerators for deep learning. In: IEEE/ACM international conference on computer-aided design (ICCAD)

Liu R, Peng X, Sun X, Khwa WS, Si X, Chen JJ, Li JF, Chang MF, Yu S (2018) Parallelizing SRAM arrays with customized bit-cell for binary neural networks. In: ACM/IEEE design automation conference (DAC)

Liu Q, Gao B, Yao P, Wu D, Chen J, Pang Y, Zhang W, Liao Y, Xue CX, Chen WH, Tang J (2020) A fully integrated analog ReRAM based 78.4 TOPS/W compute-in-memory chip with fully parallel MAC computing. In: IEEE international solid-state circuits conference (ISSCC)

Long Y, She X, Mukhopadhyay S (2019) Design of reliable DNN accelerator with un-reliable ReRAM. In: Design, Automation & Test in Europe Conference & Exhibition (DATE)

Lue HT, Hsu PK, Wei ML, Yeh TH, Du PY, Chen WC, Wang KC, Lu CY (2019) Optimal design methods to transform 3D NAND flash into a high-density, high-bandwidth and low-power nonvolatile computing in memory (nvCIM) accelerator for deep-learning neural networks (DNN). In: IEEE international electron devices meeting (IEDM)

Luo Y, Peng X, Hatcher R, Rakshit T, Kittl J, Rodder MS, Seo JS, Yu S (2020) A variation robust inference engine based on STT-MRAM with parallel read-out. In: IEEE international symposium on circuits and systems (ISCAS)

Mikolajick T, Schroeder U, Slesazeck S (2020) The past, the present, and the future of ferroelectric memories. IEEE Transactions on Electron Devices 67(4):1434–1443

Peng X, Huang S, Luo Y, Sun X, Yu S (2019) DNN+NeuroSim: an end-to-end benchmarking framework for compute-in-memory accelerators with versatile device technologies. In: IEEE international electron devices meeting (IEDM)

Peng X, Liu R, Yu S (2020) Optimizing weight mapping and data flow for convolutional neural networks on processing-in-memory architectures. IEEE Trans Circuits Syst I Regul Pap 67(4):1333–1343

Presley RK, Haggard RL (1994) A fixed point implementation of the backpropagation learning algorithm. In: Proceedings of SOUTHEASTCON

Rastegari M, Ordonez V, Redmon J, Farhadi A (2016) XNOR-net: ImageNet classification using binary convolutional neural networks. In: European conference on computer vision (ECCV)

Ronen R, Eliahu A, Leitersdorf O, Peled N, Korgaonkar K, Chattopadhyay A, Perach B, Kvatinsky S (2022) The bitlet model: a parameterized analytical model to compare PIM and CPU systems. ACM J Emerg Technol Comput Syst 18(2):1–29

Shafiee A, Nag A, Muralimanohar N, Balasubramonian R, Strachan JP, Hu M, Williams RS, Srikumar V (2016) ISAAC: a convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In: ACM/IEEE international symposium on computer architecture (ISCA), vol 44, p 14

Shim W, Yu S (2021) Technological design of 3D NAND based compute-in-memory architecture for GB-scale deep neural network. IEEE Electron Device Lett 42(2):160–163

Si X et al (2020) A 28nm 64Kb 6T SRAM computing-in-memory macro with 8b MAC operation for AI edge chips. In: IEEE international solid-state circuits conference (ISSCC)

Simonyan K, Zisserman A (2015) Very deep convolutional networks for large-scale image recognition. In: International conference on learning representations (ICLR)

Song L, Qian X, Li H, Chen Y (2017) PipeLayer: a pipelined RRAM-based accelerator for deep learning. In: IEEE international symposium on high performance computer architecture (HPCA)

Song T, Jung J, Rim W, Kim H, Kim Y, Park C, Do J, Park S, Cho S, Jung H, Kwon B, Choi H-S, Choi J, Yoon JS (2018) A 7nm FinFET SRAM using EUV lithography with dual write-driver-assist circuitry for low-voltage applications. In: IEEE international solid-state circuits conference (ISSCC)

Su JW et al (2020) A 28nm 64kb inference-training two-way transpose multibit 6T SRAM compute-in-memory macro for AI edge chips. In: IEEE international solid-state circuits conference (ISSCC)

Sun X, Yu S (2019) Impact of non-ideal characteristics of resistive synaptic devices on implementing convolutional neural networks. IEEE J Emerg Sel Top Circuits Syst 9(3):570–579

Sun X, Wang P, Ni K, Datta S, Yu S (2018) Exploiting hybrid precision for training and inference: a 2T-1FeFET based analog synaptic weight cell. In: IEEE international electron devices meeting (IEDM)

Ueyoshi K, Ando K, Hirose K, Takamaeda-Yamazaki S, Kadomoto J, Miyata T, Hamada M, Kuroda T, Motomura M (2018) QUEST: a 7.49 TOPS multi-purpose log-quantized DNN inference engine stacked on 96MB 3D SRAM using inductive-coupling technology in 40nm CMOS. In: IEEE international solid-state circuits conference (ISSCC)

Vanhoucke V, Senior A, Mao MZ (2011) Improving the speed of neural networks on CPUs. In: Conference on neural information processing systems (NIPS)

Wang J, Wang X, Eckert C, Subramaniyan A, Das R, Blaauw D, Sylvester D (2019) A 28-nm compute SRAM with bit-serial logic/arithmetic operations for programmable in-memory vector computing. IEEE J Solid State Circuits 55(1):76–86

Wilton Steven JE, Jouppi NP (1996) CACTI: an enhanced cache access and cycle time model. IEEE J Solid State Circuits 31(5):677–688

Wu S, Li G, Chen F, Shi L (2018) Training and inference with integers in deep neural networks. In: International conference on learning representations (ICLR)

Xia L, Li B, Tang T, Gu P, Yin X, Huangfu W, Chen PY, Yu S, Cao Y, Wang Y, Xie Y, Yang H (2016) MNSIM: simulation platform for memristor-based neuromorphic computing system. In: IEEE/ACM design automation and test in Europe Conference & Exhibition (DATE)

Xue CX et al (2019) A 1Mb multibit ReRAM computing-in-memory macro with 14.6ns parallel MAC computing time for CNN-based AI edge processors. In: IEEE international solid-state circuits conference (ISSCC)

Yeap G, Lin SS, Chen YM, Shang HL, Wang PW, Lin HC, Peng YC, Sheu JY, Wang M, Chen X, Yang BR (2019) 5nm CMOS production technology platform featuring full-fledged EUV, and high mobility channel FinFETs with densest 0.021 μm2 SRAM cells for mobile SoC and high performance computing applications. In: IEEE international electron devices meeting (IEDM)

Yin S, Jiang Z, Seo JS, Seok M (2020a) XNOR-SRAM: in-memory computing SRAM macro for binary/ternary deep neural networks. IEEE J Solid State Circuits 55(6):1733–1743

Yin S, Sun X, Yu S, Seo JS (2020b) High-throughput in-memory computing for binary deep neural networks with monolithically integrated RRAM and 90nm CMOS. IEEE Trans Electron Devices 67(10):4185–4192

Yoon JH, Chang M, Khwa WS, Chih YD, Chang MF, Raychowdhury A (2021) A 40nm 64Kb 56.67 TOPS/W read-disturb-tolerant compute-in-memory/digital RRAM macro with active-feedback-based read and in-situ write verification. In: IEEE international solid-state circuits conference (ISSCC)

Yu S (2018) Neuro-inspired computing with emerging nonvolatile memorys. Proc IEEE 106(2):260–285

Yu S, Chen PY, Cao Y, Xia L, Wang Y, Wu H (2015) Scaling-up resistive synaptic arrays for neuro-inspired architecture: challenges and prospect. In: IEEE international electron devices meeting (IEDM)

Zhou S, Wu Y, Ni Z, Zhou X, Wen H, Zou Y (2016) Dorefa-net: training low bitwidth convolutional neural networks with low bitwidth gradients. arXiv preprint arXiv:1606.06160

Zhu R, Tang Z, Ye S, Huang Q, Guo L, Chang S (2021) Memristor-based image enhancement: high efciency and robustness. IEEE Trans Electron Devices 68(2):602–609

Design Automation Techniques for Microfluidic Biochips

20

Xing Huang, Tung-Che Liang, Zhanwei Zhong, Tsung-Yi Ho, and Krishnendu Chakrabarty

Contents

Introduction . . . 688
Flow-Based Microfluidic Biochips . . . 690
Design Tasks for FBMBs . . . 691
Design Automation for FBMBs . . . 693
Synthesis Methods for the Flow Layer . . . 693
Synthesis Methods for the Control Layer . . . 696
Synthesis Methods for the Codesign of the Control and Flow Layers . . . 699
Digital Microfluidic Biochips . . . 701
Technology Platforms and Applications . . . 701
Synthesis Methods . . . 704
MEDA Biochips . . . 707
Hardware Implementation . . . 708
MEDA Evolution . . . 711
Synthesis Methods . . . 712
Conclusion . . . 718
References . . . 719

X. Huang
School of Computer Science, Northwestern Polytechnical University, Xi'an, China
e-mail: xing.huang@nwpu.edu.cn; xing.huang@tum.de

T.-C. Liang · Zhanwei Zhong
Department of Electrical and Computer Engineering, Duke University, Durham, NC, USA
e-mail: tung.che.liang@duke.edu; zzhong@marvell.com

T.-Y. Ho
Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, China
e-mail: tyho@cse.cuhk.edu.hk; tyho@cs.nthu.edu.tw

K. Chakrabarty (✉)
School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, AZ, USA
e-mail: krishnendu.chakrabarty@asu.edu; krishnendu.chakrabarty@duke.edu

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_63

Abstract

Microfluidics-based biochips enable the precise control of nanoliter volumes of biochemical samples and reagents. They combine electronics with biology and integrate various bioassay operations, such as sample preparation, analysis, separation, and detection. Compared to conventional laboratory procedures, which are cumbersome and expensive, miniaturized biochips offer the advantages of higher sensitivity, lower cost, system integration, and less likelihood of human error. Because of these advantages, microfluidic biochips are being increasingly used for DNA sequencing, point-of-care clinical diagnostics, and immunoassays. This chapter describes three mainstream microfluidic technology platforms: (1) flow-based microfluidics, (2) digital microfluidics, and (3) microelectrode-dot-array biochips. The chapter presents recent advances in computer-aided design tools for simulation, synthesis, and chip optimization. These tools target modeling and simulation, scheduling, module placement, and droplet routing. With the help of these tools, biochip users can concentrate on the development of nanoscale bioassays, leaving details of chip optimization and implementation to software tools.

Keywords

Computer-aided design (CAD) · Microfluidics · Clinical diagnostics · Lab on a chip · Synthesis

Introduction

Recent advances in microfluidics have enabled the highly integrated "lab-on-a-chip" biochips. These biochips have led to a revolution in biology and biochemistry. By performing precise handling fluid samples/reagents in nanoliter volumes, various laboratory procedures (also referred to as bioassays), e.g., point-of-care diagnosis (Sista et al. 2020) and air monitoring (Huang et al. 2020), can be executed concurrently on a coin-sized microfluidic platform. Due to a fully automated execution procedure, microfluidic biochips offer several advantages over the conventional fluidic platforms that need human interventions, including higher precision, higher reliability, and lower sample/reagent consumption.

In particular, the rapid worldwide spread and impact of the COVID-19 virus have created an urgent need for reliable, accurate, and affordable testing on a massive scale. Accordingly, microfluidic devices are being adopted for COVID-19 testing for their high efficiency and fast sample-to-result turnaround. These devices, undoubtedly, play a significant role in helping people return back to normal life. For example, a 3D printed microfluidic lab-on-a-chip device has been developed to concurrently detect SARS-CoV-2 RNA and anti-SARS-CoV-2 antibodies in saliva and plasma (Najjar et al. 2022).

For the past few years, commercialization of microfluidics has been one of the main drivers for researchers developing lab-on-a-chip technology. There is a tremendous potential for growth in lab-on-a-chip technologies as more discoveries are made. A summary of the current microfluidic products is presented in Table 1. Several companies have transitioned microfluidic platforms to the marketplace, each with different design paradigms and varying degrees of success.

The road toward large-scale integration which microfluidics has taken is very similar to the early development of integrated circuits. As the available on-chip resources became unmanageable, CAD tools were introduced to improve productivity, eventually creating an electronic design automation (EDA) industry for integrated circuits. For microfluidic biochips, however, this systematic design automation is yet to come. To realize a complete automation flow for biochip design and thereby promote the development of an EDA industry for microfluidics, a large amount of work on the design automation of microfluidic biochips has been carried out over the past decade (Chakrabarty et al. 2010; Huang et al. 2019a, 2021a,b; Ibrahim et al. 2018a; Liu et al. 2021; Tseng et al. 2013). With these pioneering research, on the one hand, design tasks can be offloaded from researchers and engineers in biology and biochemistry. On the other hand, new chip architectures are explored automatically to open new doors to designers to meet requirements from future large-scale biological experiments and medical diagnoses. In this chapter, the authors describe three prominent microfluidic architectures: flow-based microfluidic biochips, digital microfluidic biochips, and microelectrode-dot-array (MEDA) biochips. The authors also describe computer-aided design (CAD) tools for the automated synthesis and optimization of biochips from bioassay protocols. Recent advances in modeling and simulation, fluidic-operation scheduling, module placement, physical design, and dynamic reconfiguration are also presented.

Table 1 Microfluidics-based biochip (lab-on-a-chip) products in the marketplace

Company	Product	Technology	Description	Applications
Fluidigm (Fluidigm 2020)	BioMark	Flow-based	Single-cell genomics	Genotyping, mRNA expression profiling
10x Genomics (10xgenomics 2020)	Chromium	Droplet-based	Single-cell genomics	Cell biology, cancer, immunology
NOWDiagnostics (NOWDiagnostics 2020)	ADEXUSDx	Paper-based	Point-of-care diagnosis	Testing for pregnancy, HIV, cardiac arrest
Baebies (FDA 2020)	Seeker	Digital	Newborn screening	Diagnosing diseases in newborns
Hengxin Bio (Hengxin bio 2020)	DBS-qPCR	Digital	Molecular diagnostic	Preparing libraries for next-generation sequencing
GenMark Dx (Genmark dx 2020)	ePlex	Digital	Molecular diagnostic	Identifying antibiotic resistance genes

Flow-Based Microfluidic Biochips

The structure of flow-based microfluidic biochips (FBMBs), as illustrated in Fig. 1a, is composed of two separate physical layers. Each layer consists of a micrometerscale channel network. Microchannels in the flow layer, also known as flow channels, are constructed on top of the substrate. Fluid samples and reagents can be injected into the chip from flow ports and transported precisely between different locations on the chip using the flow channels. Microchannels in the control layer, also known as control channels, are connected to external pressure sources through control ports to conduct air pressure to the overlapping area of the two channels, where a flexible membrane fabricated using elastomer material [polydimethylsiloxane (PDMS)] is deployed. When a control channel is pressurized with air, it pushes the membrane down into the flow channel, thus turning off fluid flow. In contrary, the membrane is restored back to its original position once the air pressure is released, essentially forming a valve as shown in Fig. 1b.

By combining valves in different manners, complex microfluidic devices such as mixers and peristaltic pumps can be constructed. Figure 1c shows the structure of a rotary mixer, where nine valves are used to realize its functional actions. For example, by closing a_5–a_6 and opening the remaining valves, a fluid sample can be loaded into the upper half of the mixer. Once the fluid samples that need to be mixed are loaded into the annular channel, the mixer can be sealed by closing a_1 and a_2, and the mixing operation can be started by actuating the peristaltic valves a_7–a_9 sequentially at a sufficiently high frequency (∼100 Hz). Accordingly, by integrating necessary devices into a single chip, bioassays can be completed automatically with a pre-customized assay plan.

In particular, when valves are arranged in a regular pattern along horizontal and vertical flow channels, a fully programmable valve array (FPVA) is obtained as shown in Fig. 1d, where microfluidic devices with different shapes and sizes can be established dynamically on the chip through a software system. Similarly, valves in this architecture (solid block in red color) are controlled by air pressure sources through the control channels (narrow channels in red color). Moreover, four

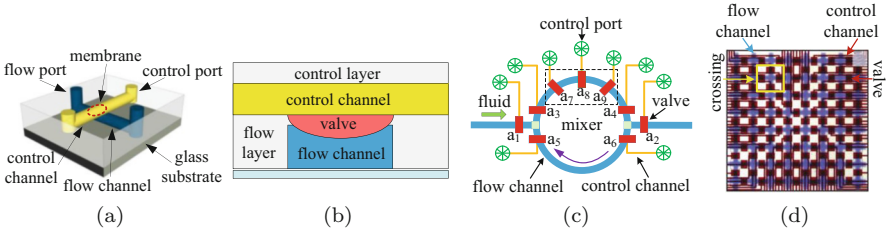


Fig. 1 (**a**) Schematic of a flow-based microfluidic biochip, (**b**) front view of (**a**), (**c**) structure of a rotary mixer, and (**d**) structure of a fully programmable valve array. (Adopted from Fidalgo and Maerkl 2011)

valves are deployed at each flow-channel (wide channels in blue color) crossing, so that variable interconnections and arbitrary channel structures can be implemented. Correspondingly, biochemical operations can be performed automatically by opening and closing a set of valves, and bioassays can be completed efficiently by configuring different functionality modules on this architecture.

As manufacturing technologies advance, the characteristic dimensions of FBMBs keep shrinking. The feature size of valves has been reduced significantly from $15\times15\,\mu m^2$ to $6\times6\,\mu m^2$ (Araci and Quake 2012). Tens of thousands of valves can be integrated into a chip smaller than a coin (Araci and Quake 2012), and the design of biochips has become much more complex to achieve desired functions. A 96 × 96 dynamic array developed by Fluidigm Inc., for example, can run 9,216 parallel polymerase chain reactions (PCR), but it requires more than 25000 valves to realize various fluidic manipulations (Perkel 2008). Consequently, traditional manual design that suffers from the drawbacks of being time-consuming and error-prone is not suitable anymore. The lack of CAD tools not only prolongs the development cycle of microfluidic products but also hinders their large-scale integration. New challenges are calling for an automatic design and integration solution.

Design Tasks for FBMBs

The automation flow of FBMBs includes several design tasks, e.g., allocating necessary devices for bioassay execution and chip control, computing exact on-chip locations for these devices, constructing efficient connections among them, etc. The results of these tasks determine the chip architecture and thus the final chip performance. In this section, the authors introduce the design tasks of architectural synthesis in both control and flow layers.

Architecture Design of the Flow Layer

The architecture design in the flow layer is performed based on a given bioassay. The protocol of an assay, also called sequencing graph, is usually modeled as a directed acyclic graph $G(O, E)$. A vertex $o_i \in O$ represents a biochemical operation, e.g., mixing and detecting, and is associated with a parameter indicating its duration. An edge $e_{i,j} \in E$ defines the dependency between operations, i.e., the resulting fluid of o_i is an input fluid of o_j. For example, Fig. 2a shows the sequencing graph of the mixing phase of PCR, where seven mixing operations (o_1–o_7) are performed to generate copies of a specific DNA sequence. Moreover, to realize the execution of these operations, a device library D is provided, where each device is modeled as a rectilinear box with input/output port on its edges.

Given the sequencing graph of an assay and a device library, the architectural synthesis in the flow layer consists of two major stages: high-level synthesis and physical design.

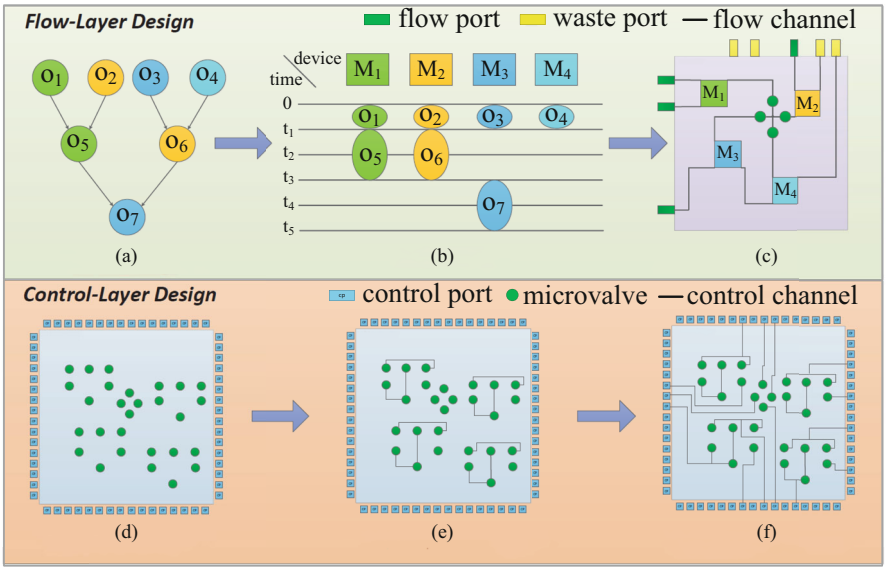


Fig. 2 Architectural synthesis framework of FBMBs. (Adopted from Yao et al. 2015). (**a**) Sequencing graph of PCR, (**b**) binding and scheduling scheme, (**c**) placement and routing solution in the flow layer, (**d**) positions of valves, (**e**) valve-addressing solution, and (**f**) routing solution in the control layer

- *High-level synthesis*: The major goals of this stage include: (i) find a binding solution $\phi : O \rightarrow D$ such that every operation in the sequencing graph is bound to a specific device to execute and (ii) compute a scheduling scheme such that all the operations can be completed efficiently while satisfying the specified dependencies. Fig. 2b shows a binding and scheduling scheme of the PCR described in Fig. 2a, where 4 mixers are allocated to execute operations. For example, operations o_1–o_4 are bound to devices M_1–M_4, respectively. They are executed concurrently at the time interval between 0 and t_1. Moreover, the complete assay is finished at time point t_5 after completing the execution of o_7 on device M_3.
- *Physical design*: With the generated binding and scheduling scheme, physical design is performed to assign the allocated devices to exact locations on the chip while establishing efficient connections among them. Typical optimization objectives in this stage include short length of flow channels, small chip area, few channel crossings, etc. For example, Fig. 2c shows a chip layout corresponding to the scheduling described in Fig. 2b, where flow ports and waste ports are placed at the boundary of the chip to inject fluid samples and recover waste fluids, respectively. In addition, the total length of flow channels and the number of channel crossings have been minimized to reduce fabrication cost of the chip.

Architecture Design of the Control Layer

After the flow-layer design is completed, the next task is to design and deploy an efficient control system, so that the functional actions required during the execution of bioassays, e.g., fluid transportation and device control, can be implemented with an accurate control of microvalves. Accordingly, the major goal of control-layer design is to establish efficient connections between valves and control ports through the following two steps: valve addressing and control-channel routing.

- *Valve addressing*: Since the density of valves integrated into a single biochip has increased significantly over the past decade – up to 1 million valves/cm^2 (Araci and Quake 2012) – this leads to a rapid increase in the number of control ports. These ports occupy extra chip area and need to be connected to external pressure sources, thus increasing the fabrication cost and complexity of the control system significantly. To solve this problem, valve addressing is performed to group valves that can be actuated in a compatible manner while assigning them a shared control port. Figure 2d shows the positions of valves in the chip layout described in Fig. 2c. By computing the switching patterns of these valves at each time step, the compatibility among them can be identified. In the grouping solution shown in Fig. 2e, valves that are connected together can be actuated by the same control port.
- *Control-channel routing*: With an optimized valve-addressing solution, routing of control channels is then performed to connect valves in the same group to their control port. Figure 2f shows the routing result corresponding to Fig. 2e. In this architecture, control ports are placed at the edges of the control layer to reduce design complexity. Note that control ports can actually be placed anywhere on the chip. Optimization goals in this stage typically include short length of control channels, synchronization of valve actuation, short-pattern setup time of control signals, etc.

Design Automation for FBMBs

In this section, the corresponding design methods for both control and flow layers are discussed.

Synthesis Methods for the Flow Layer

Many design automation methods have recently been proposed for the flow-layer architectural synthesis of FBMBs (Huang et al. 2019a, 2021a,b; Ibrahim et al. 2018a; Liu et al. 2021; Tseng et al. 2013; Huang et al. 2022). These synthesis tools adopt various optimization techniques, including integer linear programming (ILP), particle swarm optimization (PSO), A*-search algorithm, etc., to systematically solve the problems described in section "Architecture Design of the Flow Layer," thus generating biochip architectures with both high efficiency and low cost.

The work in Tseng et al. (2013) presents a top-down synthesis methodology to generate optimized biochip architectures while minimizing the total amount of valve-switching. In this method, a set-based minimum cost maximum flow algorithm is proposed to generate binding and scheduling schemes with minimized completion times of bioassays. Moreover, an incremental cluster expansion method and the Dijkstra shortest-path algorithm are adopted to deal with the device placement and flow-channel routing problems, respectively. Since valve-switching reduction is considered in the complete synthesis flow, unnecessary switching of valves can be avoided, thus improving the reliability of biochips significantly.

The work in Huang et al. (2019a) proposes a synthesis flow called *MiniControl* to generate chip architectures under strict constraints of control ports. The key contribution of this work is that control-port minimization is considered systematically during the complete flow-layer design for the first time. MiniControl incorporates the constraints of control ports into a PSO-based high-level synthesis framework, where interactive strategies such as local and global experience perceptions are implemented to iteratively update the positions of particles, thereby exploring the whole search space to find the scheduling with minimized requirements of control ports. During physical design, control-port minimization is integrated into an A*-search-based channel routing algorithm. By synchronizing the switching activities of valves in channel crossings with existing on-chip valves, the already allocated control ports can be further shared by these new introduced valves.

To further improve the performance of FBMBs, the concept of distributed channel-storage architecture is put forward by removing the conventional dedicated storages from biochips (Tseng et al. 2015). The latter, as illustrated in Fig. 3a, suffers

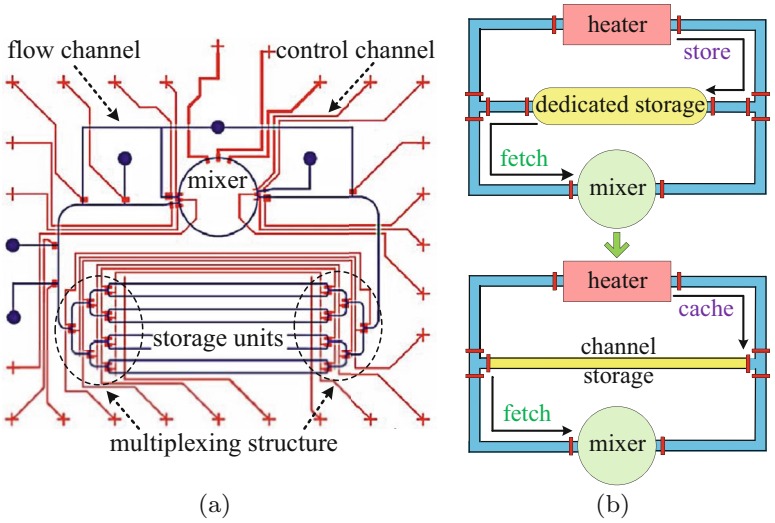


Fig. 3 Illustration of distributed channel-storage architecture. (**a**) Biochip with a dedicated storage. (Adopted from Urbanski et al. 2006) and (**b**) concept of channel storage

from several limitations, e.g., constrained capacity, fixed position, and large area occupation. Moreover, the multiplexing structure in Fig. 3a allows only one fluid to enter/leave the storage at a time, thus limiting its access bandwidth. In contrast, fluids can be cached directly in flow channels in a channel-storage architecture as illustrated in Fig. 3b. This can be viewed as distributing the storage units in a dedicated storage to the chip plane, leading to an "on-the-spot" fluid store/fetch with higher efficiency. Accordingly, Liu et al. (2021) presents an ILP-based methodology to automatically generate chip layouts with channel storage. In Huang et al. (2021a), a fast heuristic is proposed to efficiently compute optimized channel-storage architectures while removing the contaminants left behind in channels/devices during the execution of bioassays. In particular, the work in Ibrahim et al. (2018a) presents an efficient synthesis flow called *CoSyn* to generate a hybrid microfluidic platform that enables complete single-cell analysis on a heterogeneous pool of cells.

More recently, a synthesis flow named *PathDriver+* has been proposed by taking the actual fluid manipulations into account (Huang et al. 2021b). For example, when loading fluid samples into a device, to ensure correct execution of the operation, the air already present should be completely pushed out of the device. Meanwhile, the air used for driving the movement of fluids should avoid entering into the device. This constraint, consequently, requires a strict volume control of input fluids. Figure 4 describes an example of failure in volume control, where the two fluids that need to be mixed are separated due to insufficient input volumes. In contrast, in Fig. 5, volumes of the two input fluids are increased to ensure that no extra air is kept inside the mixer, leading to excess fluids left behind in flow channels. Accordingly, PathDriver formulates the constraints above into an ILP model and constructs flow paths for both fluid transportation and excess fluid removal, leading to chip architectures with volume management.

Besides the complete synthesis flows above, there are a number of automation solutions for solving the local optimizations in the flow-layer design (Lin et al. 2014; Yang et al. 2018; Wang et al. 2016; Grimmer et al. 2017; Crites et al. 2017; Huang et al. 2019b; Li et al. 2016; Lai et al. 2018). For example, the methods in Lin et al. (2014) and Yang et al. (2018) are proposed to deal specifically with the channel routing problem, so that the total length of flow channels can be minimized. Specifically, Lin et al. (2014) adopts the obstacle-avoiding rectilinear Steiner tree model to construct a flow-channel network with minimized total channel length. Similar to the X-/octilinear architecture implemented in integrated circuits, in Yang et al. (2018), the traditional Manhattan channels with 90° bends are replaced by a routing strategy with any-angle bends, thus fundamentally increasing the routing flexibility. The methods in Wang et al. (2016), Grimmer et al. (2017),

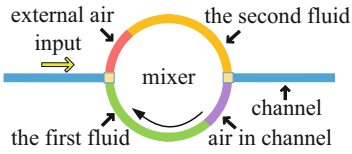


Fig. 4 Failure of fluid volume management when fluids are loaded into a rotary mixer. (Adopted from Huang et al. 2021b)

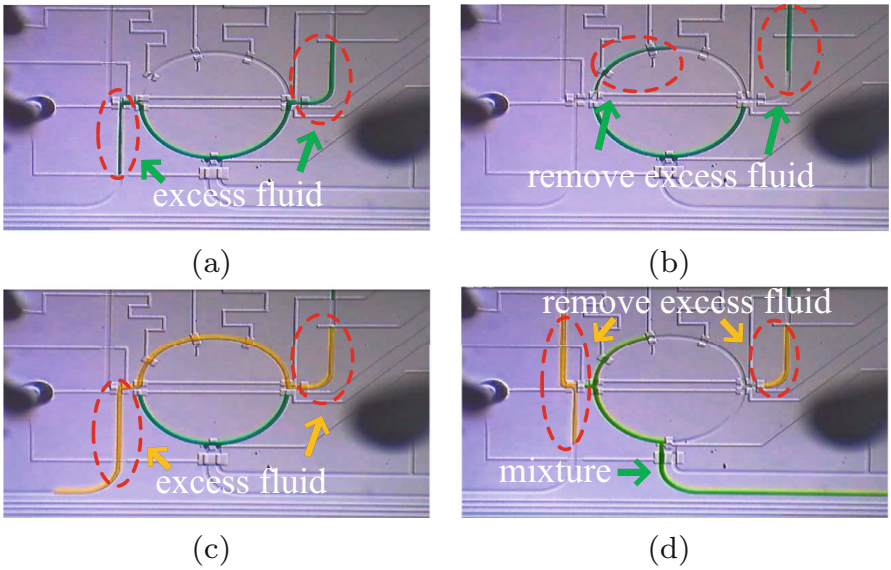


Fig. 5 Snapshots of fluid manipulations when performing the mixing operation. (Adopted from Urbanski et al. 2007). (**a**) Loading the first fluid flow, (**b**) removing the excess fluids in (**a**), (**c**) loading the second fluid flow and starting the mixing operation, and (**d**) recovering the resulting mixture and removing the excess fluids in (**c**)

Crites et al. (2017), and Huang et al. (2019b) are put forward to find optimized physical design solutions, so that key indicators such as chip area, channel length, and channel crossings can be minimized simultaneously. The method in Li et al. (2016) is proposed to compute optimized binding and scheduling schemes such that the completion time of bioassays is minimized. The synthesis methods in Lai et al. (2018) are proposed to solve the dynamic mapping and fluidic routing problems in FPVA biochips. Moreover, in Tseng et al. (2016), a valve-role-changing concept is proposed to improve the reliability of FPVA biochips, including the following two major techniques: (1) valve actuation activities are distributed evenly on the chip, and (2) the largest number of valve actuations is minimized.

Qualitative comparisons among different flow-layer architecture design methods are presented in Table 2.

Synthesis Methods for the Control Layer

To effectively implement the functionalities in the flow layer, considerable effort has been directed toward the construction of efficient control systems (Hu et al. 2014; Minhass et al. 2013; Schneider et al. 2018; Wang et al. 2017; Wu et al. 2018; Yao et al. 2015; Zhu et al. 2019). Existing CAD techniques for the control-layer

Table 2 Comparison among flow-layer architecture design methods

Method		Function[a]	Feature	Runtime
Complete synthesis flow	Tseng (Tseng et al. 2013)	H&P&R	Reliability-oriented	medium
	Huang (Huang et al. 2019a)	H&P&R	Control-port constrained	Fast
	Liu (Liu et al. 2021)	H&P&R	Distributed channel storage	Slow
	Huang (Huang et al. 2021a)	H&P&R	Distributed channel storage & washing	Fast
	Huang (Huang et al. 2021b)	H&P&R	Volume management & flow-path planning	Slow
	Ibrahim (Ibrahim et al. 2018a)	H&P&R	Hybrid microfluidic platform for single-cell genomics	Fast
Local design	Lin (Lin et al. 2014)	R	Steiner-tree oriented	Fast
	Yang (Yang et al. 2018)	R	Any angle routing	Fast
	Wang (Wang et al. 2016)	P&R	Place-route codesign	Medium
	Grimmer (Grimmer et al. 2017)	P&R	Close-to-optimal	Slow
	Crites (Crites et al. 2017)	P&R	High area utilization	Fast
	Huang (Huang et al. 2019b)	P&R	Better timing behaviors	Fast
	Li (Li et al. 2016)	H	Sieve-valve-based optimization	Slow
	Lai (Lai et al. 2018)	R	Routability-driven (FPVAs)	Fast
	Tseng (Tseng et al. 2016)	P&R	Reliability-aware (FPVAs)	Slow

[a] H: high-level synthesis; P: device placement; R: flow-channel routing

architecture design can broadly be divided into two categories: place-route-based synthesis methods and control-multiplexing-based synthesis methods.

The place-route-based synthesis strictly follows the basic design flow mentioned in section "Architecture Design of the Control Layer." First, valve addressing is performed to compute the switching patterns of valves and thus divide them into multiple inner-compatible groups. Afterward, a control port is allocated to each group to generate shared control patterns. The allocated ports are then placed at appropriate locations on the control layer and connected to the valves through control channels, thus generating efficient control architectures with minimized cost.

A set of place-route-based synthesis methods have been proposed for automated control-layer design (Hu et al. 2014; Minhass et al. 2013; Schneider et al. 2018; Wu et al. 2018; Yao et al. 2015). The work in Hu et al. (2014) presents a top-down synthesis framework to generate timing-efficient control architectures. In this method,

an operation-based compatibility identification technique is employed to reduce the number of control ports. Moreover, an efficient heuristic that incrementally increases the priorities of failed routing nets is implemented to realize the placement of control ports as well as the routing of control channels. In the meantime, by reducing the propagation delay of air pressures, performance parameters, including response time of valves, pattern set time, timing skews, etc., have been optimized systematically in the constructed control architectures. In Wu et al. (2018), a synthesis methodology called *SOLAR* is proposed to generate control architectures with minimized number of control ports, total channel length, and timing skews. Besides, several methods have been proposed to deal with the subproblems in the control-layer design. For example, the methods in Minhass et al. (2013) and Schneider et al. (2018) are proposed to solve the control-port minimization problem. In Yao et al. (2015), a length-matching router is presented to address the timing-aware control-channel routing problem.

In contrast, control-multiplexing-based synthesis realizes efficient valve control by deploying a microfluidic multiplexer in the control layer. Figure 6 shows the structure of a control multiplexer with three control channels (Zhu et al. 2019), where the core input provides a pressure source that can be switched on or off. On the left, two pairs of complementary control ports x_1, $\overline{x_1}$, x_2, and $\overline{x_2}$ are used to create control patterns that specify which control channel is connected to the core input. Once a control channel is connected, the states of the flow valves connected with this channel will be changed to the same as that of the core input. For example, when high pressures are injected from ports x_1 and x_2, the leftmost channel in Fig. 6 is connected to the core input and thus updates the state of flow valve as 1. With this mechanism, n control ports can be used to multiplex up to $2^{n/2}$ control channels. Since each control channel can be used to manipulate a set of mutually compatible flow valves, the complexity of control systems can be reduced significantly.

Thus far, several synthesis flows have been proposed to deal with the control-multiplexing-based architecture design (Wang et al. 2017; Zhu et al. 2019). For example, in Zhu et al. (2019), an ILP-based method is presented to automatically

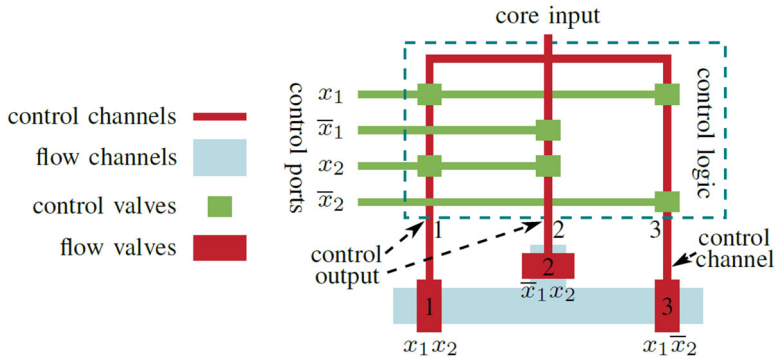


Fig. 6 Structure of a multiplexer with three control channels. (Adopted from Zhu et al. 2019)

Table 3 Comparison among control-layer architecture design methods

Method		Function[a]	Feature	Runtime
Place-route based	Hu (Hu et al. 2014)	V&P&R	Performance& cost-driven	Fast
	Wu (Wu et al. 2018)	V&P&R	Performance& cost-driven	Fast
	Minhass (Minhass et al. 2013)	V	Control-port minimization	Fast
	Schneider (Schneider et al. 2018)	V	Control-port minimization	Fast
	Yao (Yao et al. 2015)	R	Synchronization-oriented	Medium
Control multiplexing	Zhu (Zhu et al. 2019)	AD	Performance& cost-driven	Fast
	Wang (Wang et al. 2017)	LO	Reliability-oriented	Medium
	Ibrahim (Ibrahim et al. 2018b)	LO	Pin-constrained design for single-cell screening	Fast

[a] V: valve addressing; P: control-port placement; R: control-channel routing; AD: architecture design; LO: local optimization

generate control architectures supporting both multichannel switching and fault tolerance. Meanwhile, several key parameters, such as durations of channel switchings, the number of control valves (green blocks in Fig. 6), and the total length of control channels, have also been minimized accordingly. Moreover, a hybrid PSO-based flow is proposed to improve the efficiency of control-architecture design. In Wang et al. (2017), by reducing the total switching times of control valves in the multiplexer, a Hamming distance-based method is proposed to improve the reliability of the generated control architecture. More recently, the work in Ibrahim et al. (2018b) develops a synthesis tool called *Sortex* to generate pin-constrained reconfigurable biochips for single-cell screening.

Qualitative comparisons among different control-layer design methods are presented in Table 3.

Synthesis Methods for the Codesign of the Control and Flow Layers

As mentioned before, control and flow layers interact with each other through valves, which therefore implies that the control-layer design is closely related to the synthesis solution of the flow layer. For example, a chip layout with plenty of channel crossings gathered in a specific region in the flow layer would harm the routability of control channels and may even result in design failure of the control layer. The synthesis methods discussed in sections "Synthesis Methods for the Flow Layer" and "Synthesis Methods for the Control Layer," on the other hand, can

generate optimized layout solutions for either the control or flow layer, but they neglect the layer interactions to varying degrees and design each layer separately, leading to a gap between the two layers. Accordingly, another set of automation methods that handles the design of control and flow layers jointly has recently been proposed (Yao et al. 2015; Tseng et al. 2017, 2019).

The work in Yao et al. (2015) puts forward the first flow-control codesign methodology that seamlessly integrates the design stages in both layers. The core technique adopted in this method is a placement adjustment algorithm, which iteratively refines positions of devices in the flow layer based on the feedback information from control- and flow-layer routing stages. With this iterative feedback tuning, congestions in both layers can be eliminated, and the overall solution quality can be improved significantly.

In particular, a co-layout synthesis tool suite named *Columba* has recently been proposed to bridge the design gap between control and flow layers (Tseng et al. 2017, 2019). These tools have received considerable attention in both EDA and microfluidic communities, due to their advantages in dealing with large-scale designs within complex layer interactions. Major features of Columba include the following:

1. Columba provides a module library for accurately modeling various microfluidic devices with valve implementation, thus enabling layer interactions during architecture design.
2. Columba takes plaintext design specifications as inputs (see Fig. 7) and performs concurrent, planarity-guaranteed place-and-route design for both control and flow layers.
3. Columba generates AutoCAD-compatible solutions that can be directly used for mask fabrication. Figure 7 shows a fabricated biochip for kinase activity application based on the architecture solution synthesized by Columba.
4. Columba is easily accessible to a wide range of users with a web interface as illustrated in Fig. 8.

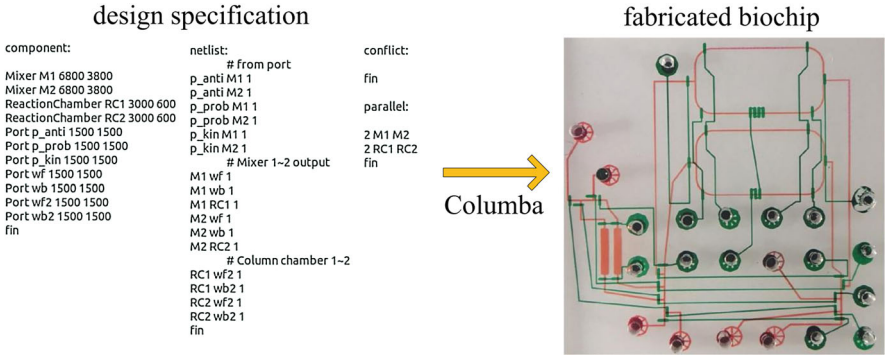


Fig. 7 Design specification of kinase activity application and the corresponding biochip synthesized by Columba. (Adopted from Tseng et al. 2017)

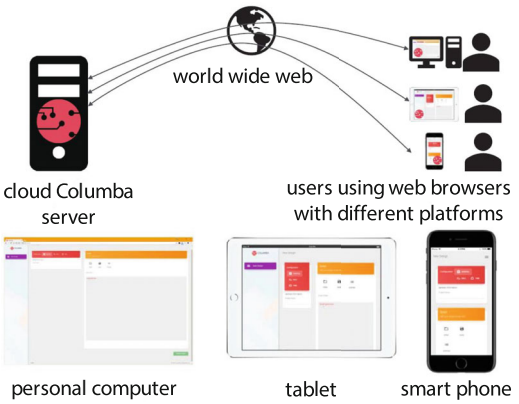


Fig. 8 Cloud Columba is easily accessible using a web browser on various portable platforms. (Adopted from Tseng et al. 2019)

With the proposed Columba tool suite, users with different background only need to upload a specification at high abstraction level describing their design requests to the cloud server, and a customized manufacturing-ready design solution will be returned automatically within minutes.

Digital Microfluidic Biochips

A digital microfluidic biochip (DMFB) is a reconfigurable lab-on-a-chip technology that is well-suited for cyberphysical integration; a DMFB allows biochemical reactions to be carried out on-demand based on a feedback sensory signal. DMFBs have therefore achieved considerable success in enabling miniaturized analysis systems for several contemporary biochemical applications. Recently, fast-turnaround tests for COVID-19 viruses based on DMFB devices have been approved by government officials and introduced to the marketplace (FDA 2021). Figure 9 shows a DMFB fabricated by Baebies.

Technology Platforms and Applications

A digital microfluidic biochip (DMFB) is composed of a two-dimensional electrode array that manipulates discrete fluid droplets of nanoliter or picoliter volumes using electrowetting on dielectric (EWOD). A unit cell in the array includes a pair of electrodes that acts as two parallel plates. The bottom plate contains a patterned array of individually controlled electrodes, and the top plate is coated with a continuous ground electrode. Figure 10a shows a DMFB, where two droplets are present on a patterned electrode array (Liang et al. 2020). When driven by a sequence of control voltages, the electrode array can perform fluidic operations, such as dispensing, mixing, and splitting. As shown in Fig. 10b, an imbalance of interfacial tension is created if an electric field is applied to only one side of the

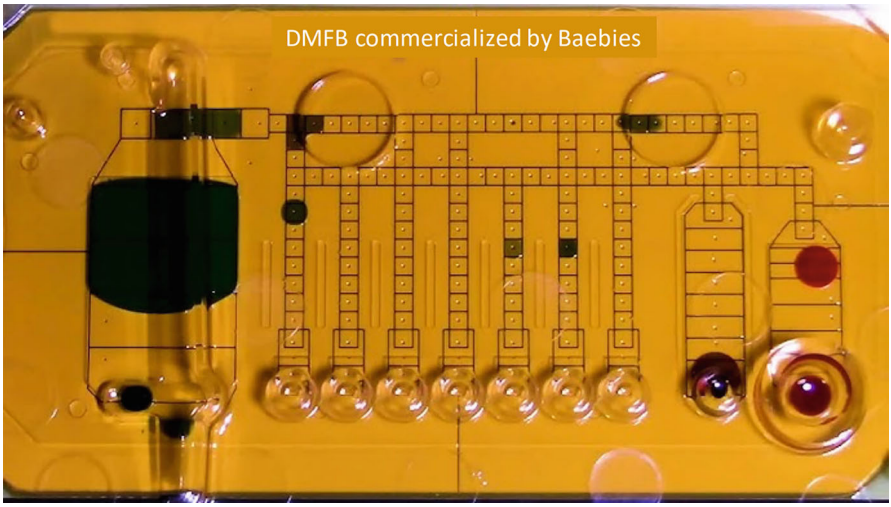


Fig. 9 DMFBs fabricated and marketed by Baebies (FDA 2016)

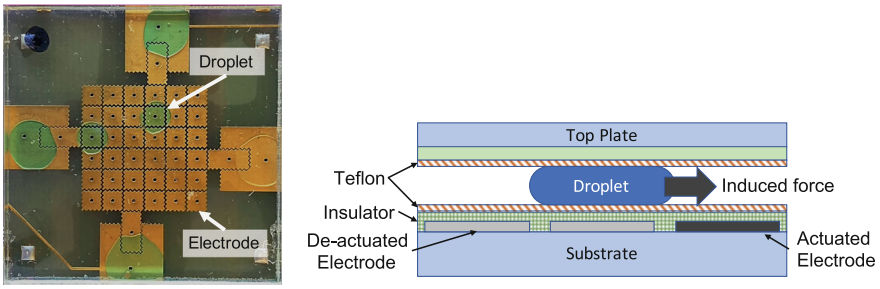


Fig. 10 (**a**) Top view of a DMFB (Liang et al. 2020). Two droplets are present on the biochip. (**b**) Illustration of the side view of a DMFB. The droplet is moved to the right using EWOD

droplet; this interfacial tension gradient forces the droplet to move toward the right. A filler medium such as silicone oil is used between the two plates to avoid fluid evaporation and reduce the likelihood of cross-contamination between samples. Today's DMFBs can also be integrated with sensors and intelligent cyberphysical control (Luo et al. 2012).

Because of the precise control over microfluidic operations, DMFBs are employed by microbiologists and biomedical engineers to seamlessly process several biomolecular reaction steps without bulky instrumentation. DMFBs have been demonstrated to handle complicated applications such as cell biology (Lamanna et al. 2020), point-of-care diagnostics (Sista et al. 2020), and air monitoring (Huang et al. 2020). Several examples are described below:

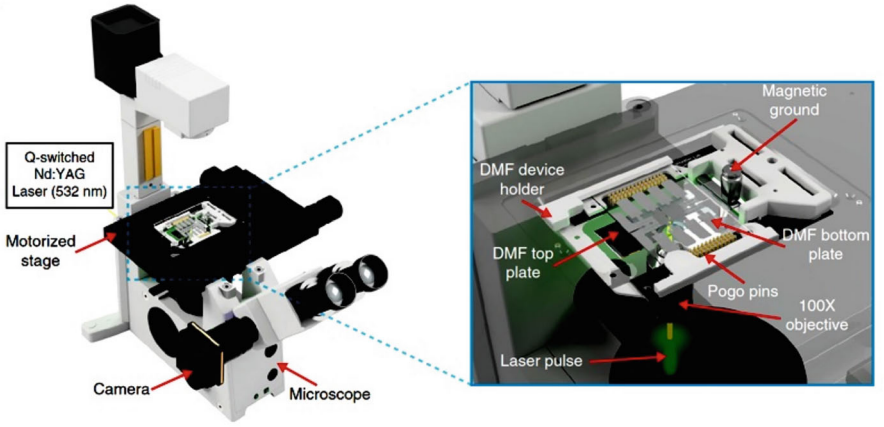


Fig. 11 DMFB platform for single-cell analysis (Lamanna et al. 2020)

Single-Cell Isolation and Analysis Single-cell analysis is the study of genomics, transcriptomics, proteomics, metabolomics, and cell-cell interactions at the single-cell level. Recently, a DMFB platform has been proposed to automate a robust and all-purpose path for single-cell analysis (Lamanna et al. 2020). This DMFB platform comprises a 2D electrode array that can be used for cell and reagent manipulation, a microscope for imaging and cell selection, and a laser module for cell lysis; see Fig. 11. The single-cell analysis on this platform has several steps: (1) To-be-studied cells are loaded to reservoirs for culturing. (2) A computer vision technology is used to identify a cell of interest. (3) Upon selection of the cell of interest, a high-energy laser pulse is delivered to the targeted cell, resulting in cell lysis and releasing its contents into solution. (4) Finally, sequencing analysis by genomics, transcriptomics, and/or proteomics is carried out on the DMFB.

Point-of-Care Diagnostics The work in Sista et al. (2020) presents a digital microfluidic testing platform for clinical assays using microliter volumes of samples. Features such as cell lysis, plasma preparation, magnetic bead washing, thermocycling, and incubation are all performed on a single DMFB without any user intervention. The automated platform is composed of a small instrument and single-use cartridges, i.e., replaceable DMFBs. This DMFB platform is the first of its kind to combine chemistry assays, immunoassays, and molecular assays together on a single system. Chemistry assay is used for neonatal hyperbilirubinemia diagnosis. Immunoassays are used in clinical laboratory testing to measure antibodies and/or antigens. Molecular diagnostics tests are commonly used for the detection of infectious diseases and genetic markers of disease risk. A rapid PCR assay, one of the molecular diagnostic tests, has been demonstrated on the platform within 5 minutes.

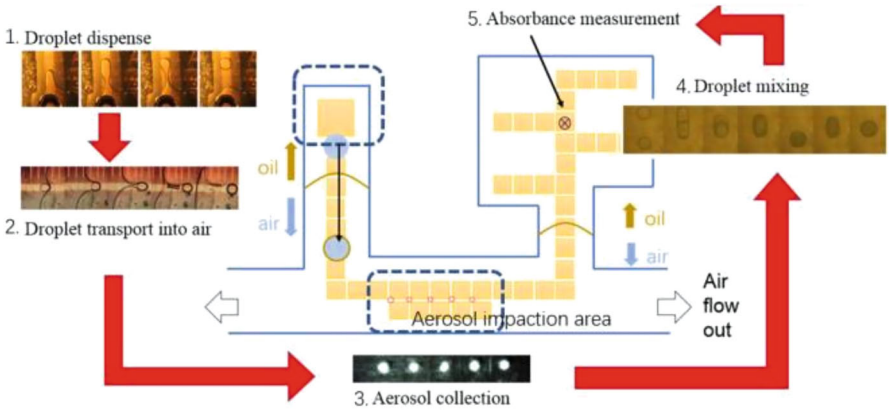


Fig. 12 DMFB system for the detection of inorganic ions in aerosols (Huang et al. 2020)

Air Monitoring Analysis of atmospheric aerosol content has been a focus of research for air pollution monitoring and climate change. Conventional aerosol measurement contains filters that capture sufficient mass of the target compounds by exposing to the air for several hours. This approach has many drawbacks such as long sampling times, high labor costs, risks of contamination, and low sensitivity within a short timescale. Recently, a digital microfluidic system has been presented, and it automates the tasks of collection, preparation, extraction, and analytical detection of inorganic ions on a single biochip (Huang et al. 2020). The overall detection procedure is shown in Fig. 12. First, a droplet is dispensed in the oil filler medium. Next, the droplet is transported to the aerosol impaction area, where the air constantly flows and the droplet captures the aerosol content. Finally, the droplet is mixed with compound-specific regents to form a colored complex, and the mixture is photometrically detected on the biochip. Because the volume of the fluid in the system is very small, the turnaround time of the detection is less than an hour.

Synthesis Methods

Considerable effort has been devoted to synthesis and optimization of digital microfluidic biochips (Chen et al. 2013; Luo et al. 2014; O'neal et al. 2017; Ricketts et al. 2006; Su and Chakrabarty 2005, 2006, 2008). A synthesis tool maps a given bioassay to the patterned electrode array of digital microfluidics, by binding fluidic operations to on-chip resources, generating an optimized schedule, and computing droplet transportation pathways. Goals for the optimization of biochip synthesis usually include (i) the minimization of bioassay completion time, (ii) the minimization of chip area, and (iii) the defect tolerance.

Scheduling and Module Placement

One of the first published methods for biochip synthesis decoupled high-level synthesis from physical design (Su and Chakrabarty 2008). Architectural-level synthesis for microfluidic biochips can be viewed as the problem of scheduling assay functions and binding them to a given number of resources so as to maximize parallelism, thereby decreasing response time. A behavioral model for a set of bioassays is first obtained based on laboratory protocols. Biochip synthesis tool then binds the operation to fluidic modules, generates the operation scheduling sequence, addresses the placement of fluidic modules, and determines the routing of droplets to achieve optimization objectives. An overall flow for high-level biochip synthesis is shown in Fig. 13.

The first task, resource binding, refers to the mapping from bioassay operations to available fluidic modules. Note that there might be several modules available for a given operation. For example, either a 2×2-array mixer or a 2×3-array mixer can be utilized for a droplet mixing operation. However, different modules for a given operation may result in different operation completion times. Similarly, a fluidic module can also be associated with multiple assay operations, which necessitates resource sharing. Once resource binding is completed, operation scheduling can be performed to determine the start and stop times of all fluidic operations, subject to both precedence and resource constraints. There are four major operation scheduling algorithms that have been proposed for biochips: modified list scheduling (MLS) (Su and Chakrabarty 2008), an optimal scheduler based on integer linear programming (ILP) (Su and Chakrabarty 2008), and two

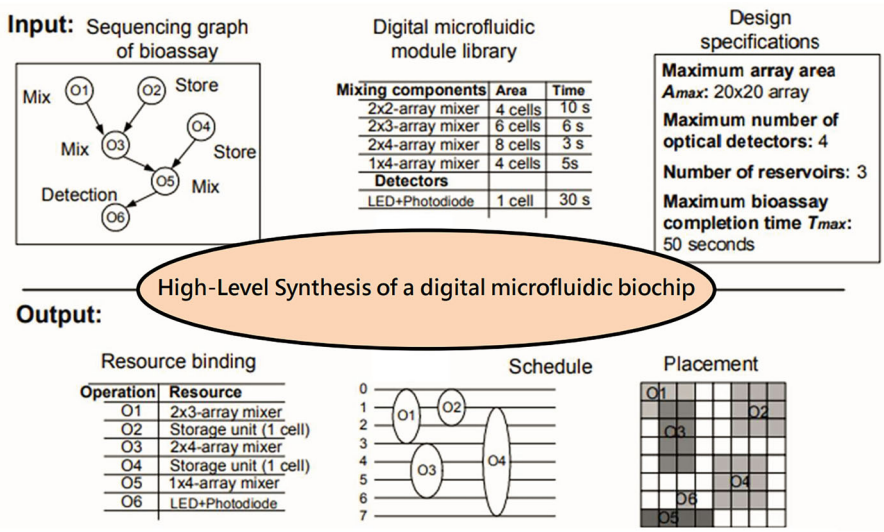


Fig. 13 An example illustrating high-level synthesis for a digital microfluidic biochip. (Adopted from Chakrabarty et al. 2010)

genetic algorithms (Ricketts et al. 2006). More recently, a resource-constrained scheduling algorithm has also been presented based on force-directed list scheduling (FDLS) (O'neal et al. 2017).

After operation scheduling is completed, the key problem is the placement of fluidic modules (e.g., different types of mixers, diluters, and splitters). Since DMFBs enable dynamic reconfiguration of fluidic modules during run-time, different modules can be placed on the same location in various time intervals. Module placement problem on a DMFB has been proven to be NP-complete (Su and Chakrabarty 2006). A simulated annealing (SA)-based heuristic approach has been proposed to efficiently solve this problem (Su and Chakrabarty 2006). Su and Chakrabarty (2005) presented a high-level synthesis technique for DMFBs by connecting operation scheduling with module placement. The work in Chen et al. (2013) proposed the first reliability-oriented non-SA placement algorithm for DMFBs, which utilizes the 3-D deferred decision-making (3D-DDM) technique to enumerate only possible placement solutions to reduce computation complexity. If the chip has not been fabricated, placement solutions can provide chip designers with information on the size of the array to be manufactured. If the chip has been fabricated, optimized module placement and area minimization free up more unit cells for defect tolerance (Chakrabarty et al. 2010).

Droplet Routing

The problem of determining paths of droplet transportation between modules is referred to as *droplet routing*. The dynamic reconfigurability inherent in digital microfluidics allows different droplet routes to share cells on the microfluidic array during different time intervals. Several systematic routing methods for digital microfluidic biochips have therefore been developed to minimize the number of cells used for droplet routing while satisfying constraints imposed by performance goals and fluidic properties (Liang et al. 2020; Huang et al. 2009; Su et al. 2006; Xu and Chakrabarty 2007).

One of the first methods for droplet routing in biochips was proposed in Su et al. (2006). The main objective in routing is to find droplet routes with minimum lengths, where route length is measured by the number of cells in the path from the starting point to the destination. For a microfluidic array of fixed size, minimum-length droplet routes lead to the minimization of the total number of cells used in droplet routing, thus freeing up more spare cells for fault tolerance.

Droplet routing should be considered in the synthesis flow for digital microfluidics, in order to generate a routable synthesized design for the availability of routing paths. The work in Xu and Chakrabarty (2007) proposed a method to incorporate droplet routability in the PRSA-based synthesis flow. This method estimates the droplet routability using two metrics. It adopts the average module distance (over all interdependent modules) as the first design metric to guarantee the routability of modules in the synthesized biochip. It also adopts the maximum module distance as the second design metric to approximate the maximum length of droplet manipulation.

Since synthesis results with high routability values are more likely to lead to simple and efficient droplet pathways, this method incorporates the above two metrics into the fitness function by a factor that can be fine-tuned according to different design specifications to control the PRSA-based procedure. Candidate designs with low routability are discarded during evolution. Thus, the synthesis procedure guarantees that the routing complexity is reduced for the synthesized biochip while meeting constraints on array size and bioassay processing time.

However, the above methods are static, and they neglect the fact that droplet transportation may fail if the electrodes associated with the routing path degrade over time. Recently, the work in Liang et al. (2020) proposed a real-time routing method that can capture the underlying health conditions of electrodes and provide reliable routing pathways. This work casts droplet transportation as a reinforcement learning problem. In the RL framework, a deep neural network is first trained to learn a reliable policy for droplet routing. Next, the network is loaded on a cyberphysical DMFB, where it can observe the health condition of the electrodes. The experimental results showed that even though electrodes on a DMFB degrade over time, the RL droplet router can learn the degradation behavior and transport droplets using only healthy electrodes. This work increases the lifespan of a biochip's utility and allows for the adaptation of a plethora of bioassays on to the DMFB platform.

MEDA Biochips

Microelectrode-dot-array (MEDA) architecture have been demonstrated in recent years to overcome the drawbacks of DMFB, which contains (1) constraints on droplet size and the inability to vary droplet volume in a fine-grained manner, (2) the lack of a sufficient number of integrated sensors for real-time droplet detection, and (3) the need for special fabrication steps and the associated reliability problems. MEDA is based on the concept of a sea of microelectrodes with an array of identical microfluidic unit components called *microelectrode cells* (MCs). Each MC consists of a microelectrode and a control/sensing circuit.

Microelectrodes can be dynamically grouped to form a micro-component that can perform different microfluidic operations on a MEDA biochip. MEDA prototypes have been fabricated using the TSMC 0.35 μm CMOS process (Lai et al. 2015a); these devices can use a power supply voltage of only 3.3 V (Lai et al. 2015a). Moreover, in contrast to conventional DMFBs, MEDA incorporates real-time capacitive sensing on every microelectrode to detect the property (droplet-property sensing) and the location (droplet-location sensing) of a droplet. The "sensing map" derived in this manner opens up the exciting opportunity of cyberphysical MEDA biochips that can dynamically respond to bioassay outcomes, perform real-time error recovery, and execute "if-then-else" protocols from biochemistry necessary to support the design of the next generation of cyberphysical systems (CPS) with integrated lab-on-a-chip sensing technology.

Hardware Implementation

On a DMFB device, tiny droplets are manipulated based on the principle of EWOD. Conventional DMFB generally contains two layers: (1) The bottom layer contains a two-dimension array of electrodes. (2) The top layer acts as a ground electrode. Between the droplet and the electrodes in the bottom layer, there is a hydrophobic layer. If there is a hydrophobic layer between the liquid and the electrode, the contact angle θ will be smaller when the electrode is activated. A smaller value of θ indicates a stronger EWOD force; therefore, this hydrophobic layer is used to increase the EWOD force. To move a droplet, the authors need to apply a high voltage to an adjacent electrode and deactivate the electrode under the droplet. Because the droplet will achieve a lower energy on the high-voltage electrode, there is a force that drags the droplet to the electrode with high voltage. By applying various voltage patterns on electrodes, droplet splitting, mixing, and dispensing operations can be implemented.

Compared with conventional DMFB biochips, MEDA biochips offer more flexibility. A conventional DMFB and a MEDA biochip are illustrated in Fig. 14. The basic unit of MEDA is a microelectrode cell (MC). It contains a microelectrode, an activation circuit, and a sensing circuit. A high voltage (HV) of around 25 V is applied to the top plate of the MEDA biochip (Lai et al. 2015b).

Based on the actuation and sensing circuit under the microelectrode, three MC functions can be implemented in an MC: droplet dragging, droplet holding, and droplet sensing. The block diagrams of the actuation and sensing circuit under each MC are shown in Fig. 15 and described below:

(1) **Droplet dragging**: When SEL = 0, ACT = 1, IN = 1, and 25 V is applied to the top plate, transistors T3 and T4 are switched on while transistors T1 and T2 are switched off. In this case, the microelectrode is directly connected to the ground. Because the top plate is at 25 V, a voltage difference appears between the top plate and the microelectrode. As a result, it generates a force that will drag droplets to this MC.
(2) **Droplet holding**: When SEL = 0, ACT = 1, IN = 0, and 25 V is applied to the top plate, all transistors (i.e., T1, T2, T3, and T4) are switched off. In this case, the voltage of the microelectrode follows that of the top plate. As a result, there is no voltage difference between the top plate and the microelectrode, and the droplet remains at this location. Since the drain of transistor T4 is connected to the microelectrode with a voltage of 25 V, it can endure a large breakdown voltage of at least 25 V (Lai et al. 2015a). For this reason, transistor T4 is intentionally fabricated as an extended drain MOSFET (EDMOS) that can endure a large breakdown voltage.
(3) **Droplet sensing**: First, the top plate is grounded, and transistors T3 and T4 are switched on to discharge the microelectrode. Next, transistors T1 and T2 are switched on, and a uA-level current charges the microelectrode to a

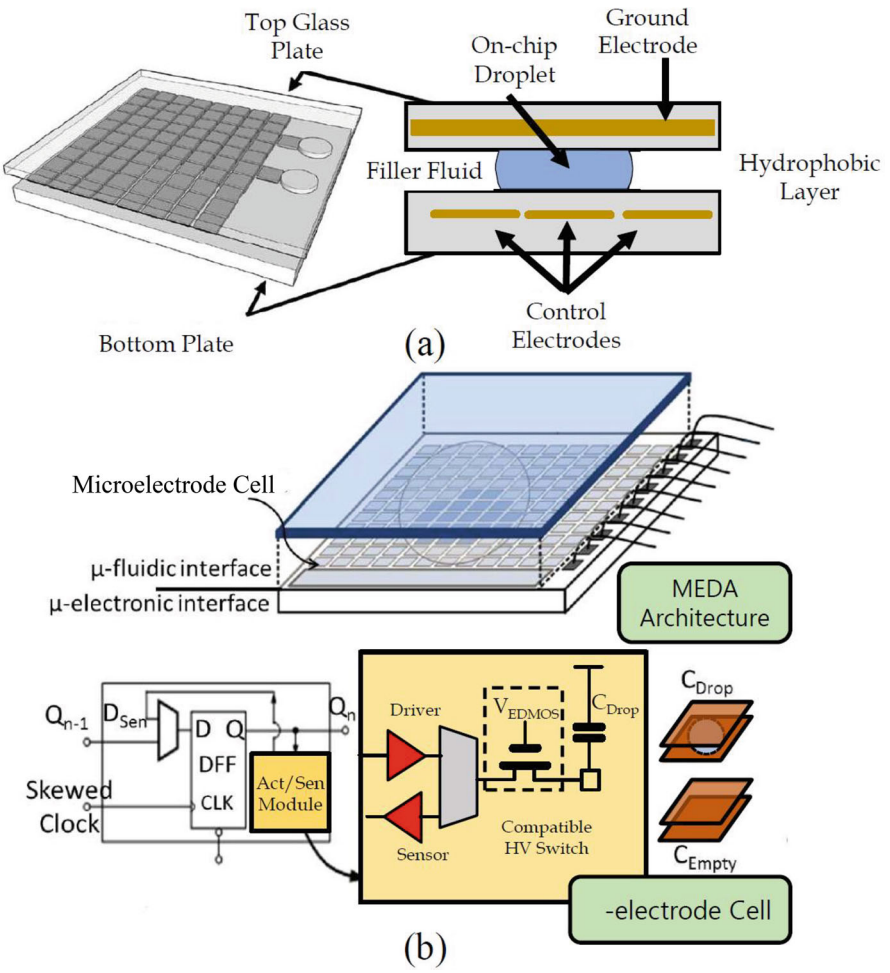


Fig. 14 The construction of (**a**) a conventional DMFB biochip and (**b**) the MEDA biochip. (Adapted from Li et al. 2018)

certain voltage. With a fixed target voltage, the charging time depends on the capacitance between the top plate and the microelectrode; therefore, the authors can detect droplet based on the small increment in charging time.

Figure 16 describes the digital signal processing of the droplet-location map. As shown in the figure, the silicone oil avoids droplet evaporation and the proposed bioprocessor has the capability to differentiate between silicone oil and DI water due to adjustable sensing threshold. Finally, droplet position and genus can be drawn by each MC sensing result.

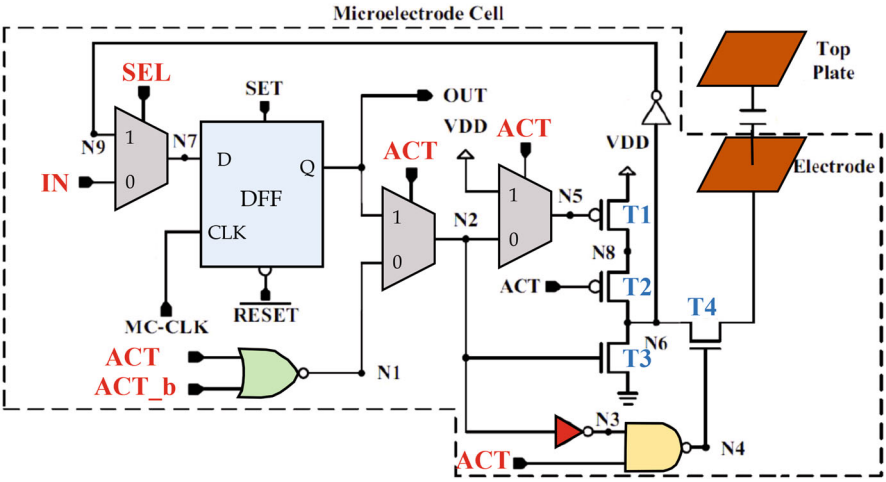


Fig. 15 Illustration of the actuation and sensing circuit under each MC. (Adopted from Lai et al. 2015b)

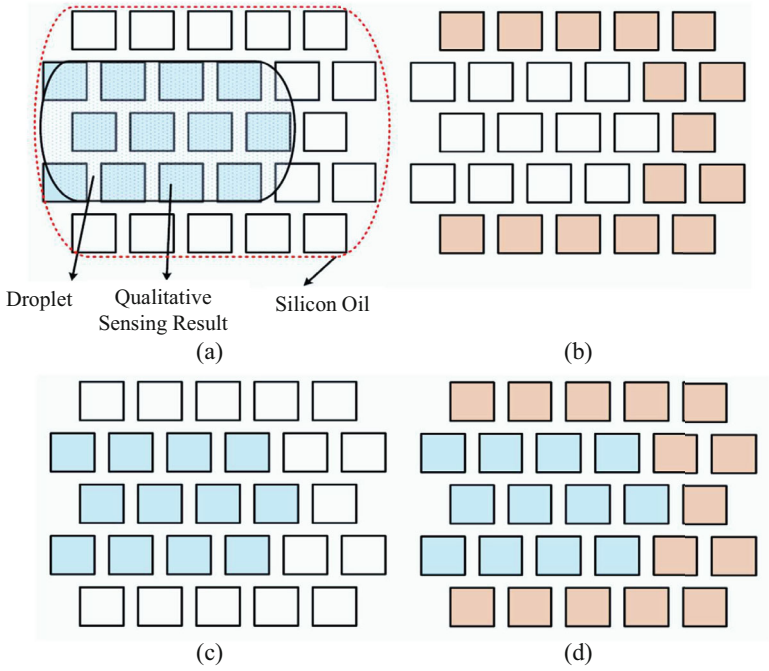


Fig. 16 (**a**) Example illustration; (**b**) silicone oil position readback; (**c**) droplet position readback; (**d**) 2D scanning image

MEDA Evolution

The first-generation MEDA biochip was fabricated in 2011 to demonstrate basic fluidic operations (Wang et al. 2011). This MEDA biochip consists of 4×12 square microelectrodes as illustrated in Fig. 17a. Each microelectrode is 100×100 mm in size with a 10 mm gap between each pair of microelectrodes. However, for this version of MEDA, the control circuit is not built in the biochip; therefore, each individually microelectrode has a wire to an outer pin to receive the controlling signal. Note that this MEDA biochip does not contain the capacitance readout circuit; therefore, droplet-location sensing is not available.

In order to overcome the above drawbacks, a second-generation MEDA biochip was fabricated in 2014 (Lai et al. 2015b). This MEDA biochip uses a standard $0.35\,u$m CMOS process, and it contains $30 \times 30 = 900$ microelectrodes; see Fig. 17b. Compared with the first-generation MEDA biochip, the major improve-

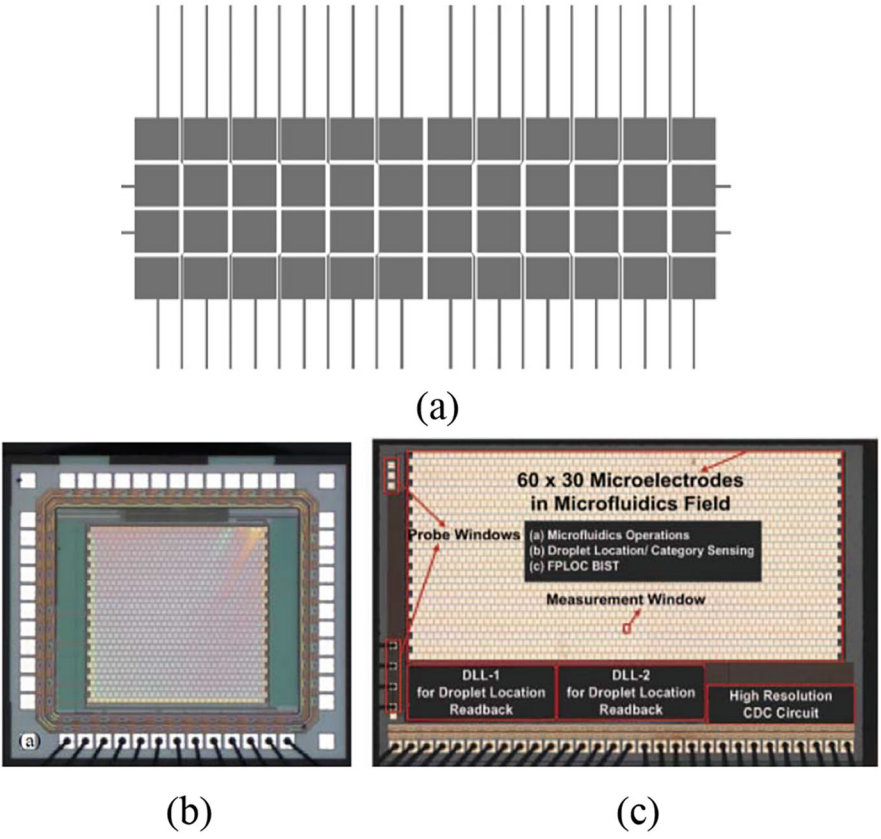


Fig. 17 Evolution of MEDA biochips: (**a**) the first generation (Wang et al. 2011), (**b**) the second generation (Lai et al. 2015b), and (**c**) the third generation (Lai et al. 2015a)

ments are as follows: (1) droplet-location sensing is included, and the "2D location map" is visible in a custom user interface, (2) droplet-property sensing (i.e., dielectric constant) can also be measured with the integrated high-sensitivity capacitance readout circuit, and (3) the controlling circuit is fully integrated, so this biochip is fully automated.

The third-generation MEDA biochip was fabricated in 2015; see Fig. 17c. Its functionality is the same as the second generation, but it significantly improves the performance of some key building blocks: (1) The on/off state of a microelectrode is controlled by an MOS switch. The breakdown voltage of the MOS switch is 25 V, while it is only 14.5 V in the second-generation MEDA biochip. With a higher breakdown voltage, the authors can apply a higher voltage to activate a microelectrode and manipulate the droplet more efficiently. (2) The droplet-location sensing circuit achieves a resolution of 1.3 fF, while the resolution is only 5 fF in the second generation. According to the analysis and experimental results reported in Lai et al. (2015a), the third-generation MEDA biochip achieves over 40% improvement in power consumption and operation time compared to the second-generation biochip.

Synthesis Methods

In this section, the authors describe the recent synthesis work specific for MEDA biochips.

Scheduling and Placement for MEDA Biochips

Even though there has been a large amount of work on high-level synthesis for conventional DMFBs, existing synthesis solutions cannot be directly utilized for MEDA biochips because of the inherent differences between conventional DMFBs and MEDA. Li et al. (2017) presents the first biochip synthesis approach that can be used for MEDA biochips. Li et al. (2017) proposed a joint synthesis flow that co-optimizes scheduling, module placement, and droplet routing for MEDA; see Fig. 18. The priority controller dynamically generates priorities for fluidic operations in a given bioassay. The scheduler, placer, and router together determine the synthesis results, i.e., start/execution time for each operation, the location of each fluidic module, and droplet pathways between start locations and end locations.

As shown in Fig. 19, the velocity model presented in Li et al. (2017) is based on the analysis of different forces imposed on the droplet, and thus it is able to calculate transportation velocities for droplets with different sizes and shapes. Next, an integer linear programming (ILP)-based reservoir placement method is proposed in Li et al. (2017). Moreover, the work in Li et al. (2017) describes a size-aware droplet router to identify droplet pathways and compute the exact droplet routing time. The routing constraints specific to MEDA biochips (i.e., static- and dynamic-constraints) are also described and experimentally validated.

Recently, an operation variation-aware module placement algorithm has been proposed for MEDA biochips (Chung et al. 2018). Due to the inherent variability

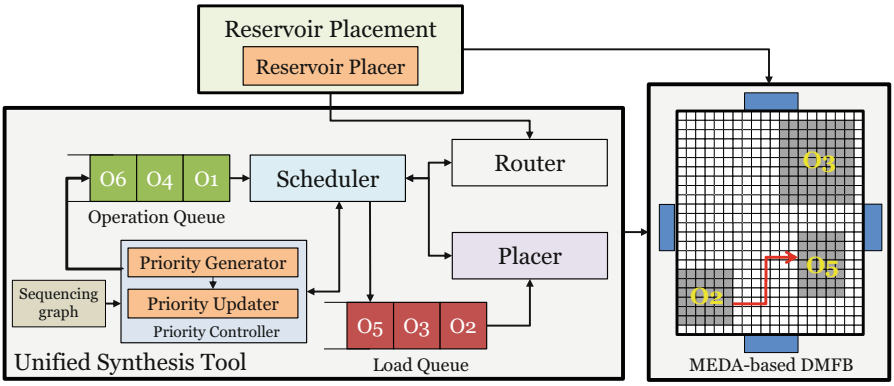


Fig. 18 Unified synthesis flow for MEDA biochips (Li et al. 2017)

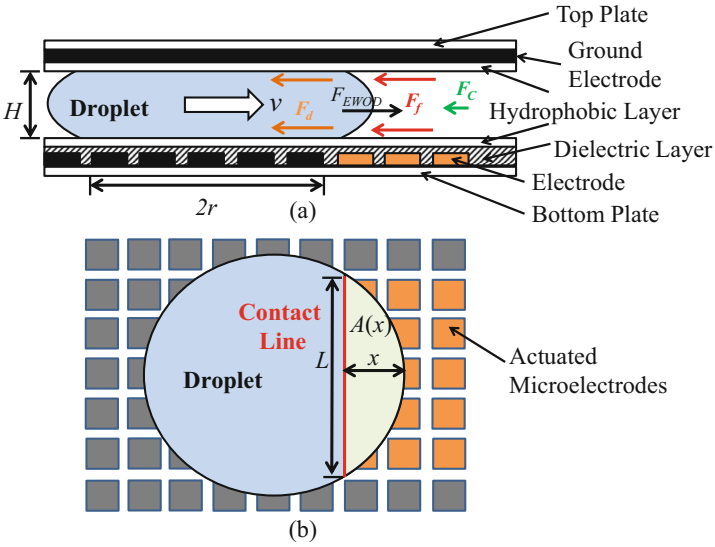


Fig. 19 A droplet undergoing transport on a MEDA biochip: (**a**) side view and (**b**) top view (Li et al. 2017)

and randomness associated with biochemical reactions, operations may be completed earlier or later during the execution of a given bioassay. The proposed algorithm in Chung et al. (2018) fully utilizes the real-time detection technique on MEDA to consider completion-time uncertainties during the execution of bioassays. In the proposed algorithm, fluidic module can be dynamically reshaped to increase the parallelism. An example is shown in Fig. 20. In Fig. 20a, there are four active modules (modules 1, 2, 4, and 6) on the chip. In order to place module 7, there are three possible solutions: reshaping module 2 (see Fig. 20b), reshaping module 4 (see Fig. 20c), and reshaping modules 1, 2, and 4 (see Fig. 20d). Once the module

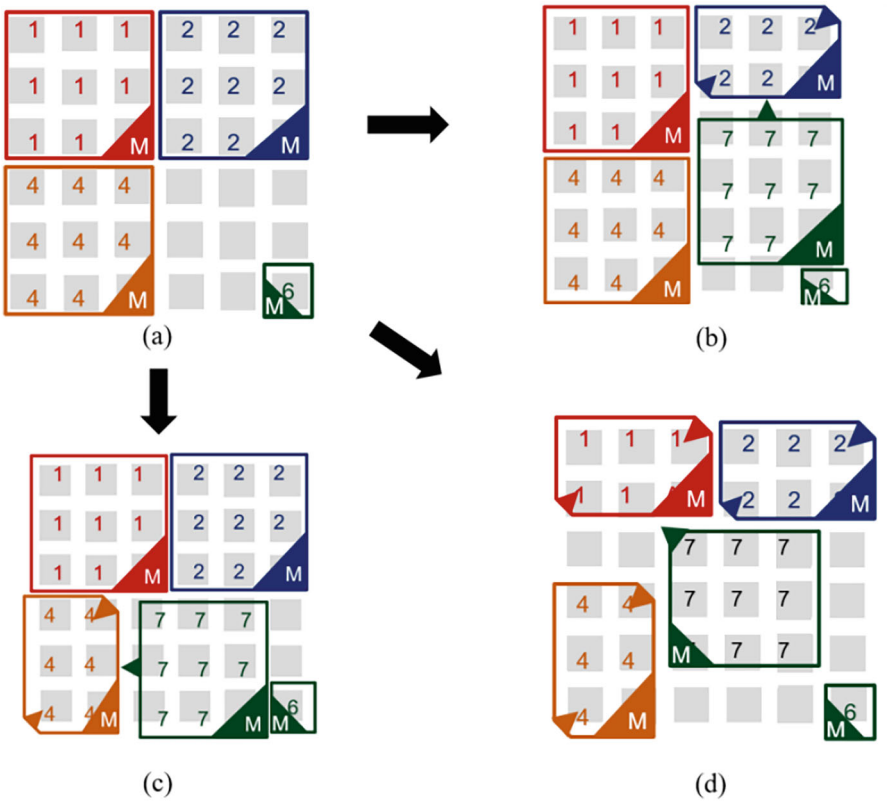


Fig. 20 Illustration of the module reshaping during placement (Chung et al. 2018)

Table 4 Comparison among different scheduling methods

Scheduling methods		Core algorithm	Runtime	Solution quality
Conventional DMFBs	Su and Chakrabarty (2008)	MLS	Fast	Medium
	Su and Chakrabarty (2008)	GA	Slow	Medium
	Ricketts et al. (2006)	HGA	Fast	High
	O'neal et al. (2017)	FDLS	Fast	High
MEDA Biochips	Li et al. (2017)	LS	Fast	Medium

is reshaped, the corresponding operation completion time may also be changed. Therefore, a dynamic scheduler was also proposed in Chung et al. (2018) to accommodate the operation execution time variation.

Qualitative comparisons between the different scheduling and module placement methods are presented in Tables 4 and 5.

Table 5 Comparison among different placement methods

Module placement methods		Core algorithm	Feature	Runtime
Conventional DMFBs	Su and Chakrabarty (2006)	SA	Fault-Tolerant	Slow
	Su and Chakrabarty (2005)	PRSA	Unified scheduling and placement	Slow
	Chen et al. (2013)	3D-DDM	Reliability-Oriented	Fast
MEDA Biochips	Li et al. (2017)	Forbidden-set-based	Unified scheduling & placement	Fast
	Chung et al. (2018)	Priority-distributed	Operation-Variation-Aware	Fast

Droplet Routing and Extension for MEDA

As described in the previous section, a key problem in biochip high-level synthesis is droplet routing. Droplet routing determines droplet routes between different fluidic modules and between modules and on-a-chip reservoirs. The dynamic reconfigurability of digital microfluidic biochips allows different droplet routes to share cells in different time intervals. Therefore, droplet routes need to be optimized to minimize the routing time as well as the route lengths. The goal of droplet routing is to determine shortest-length droplet pathways.

A set of droplet routing solutions have been proposed for conventional digital microfluidic biochips (Liang et al. 2020; Chen et al. 2013; Cho and Pan 2008; Yuh et al. 2008). However, due to the motion of multiple droplets with different sizes, droplet routing in MEDA is significantly more complicated compared with conventional DMFBs. Furthermore, there are additional fluidic operations that are feasible only on MEDA biochips, such as shape morphing and diagonal droplet motion (Li et al. 2017). These new fluidic operations, which are unique for MEDA, must also be considered by the droplet router. Accordingly, existing routing algorithms cannot be directly utilized for MEDA biochips, and droplet size-aware routers are needed for MEDA biochips.

Similar to conventional DMFBs, a minimum spacing between droplets must be maintained to prevent unexpected droplet merging during droplet routing unless the merging helps with the following operations. Fluidic constraint rules for DMFBs include both static-constraint and dynamic-constraint. Details of these constraints can be found in Li et al. (2017). Experimental demonstrations of these two constraints for MEDA biochips are shown in Figs. 21 and 22.

The first droplet routing specific for MEDA biochips was published in Chen et al. (2011). It is based on the A* algorithm, which is a graph-theoretic search and used to solve the motion planning problem, and this method is used in Chen et al. (2011) to route different sizes of droplets simultaneously. The work in

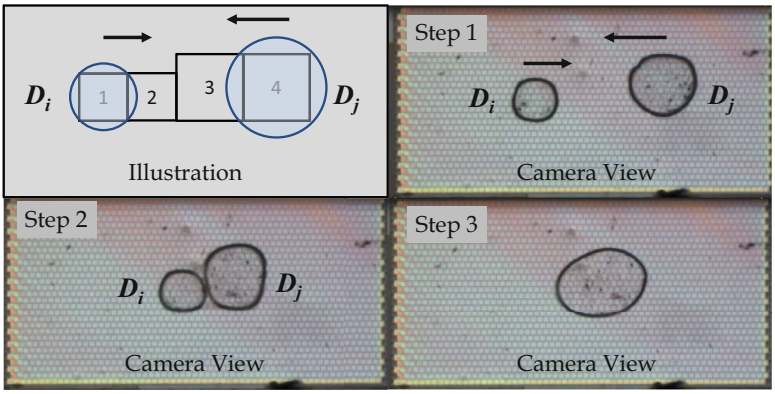


Fig. 21 Experimental verification of the static-constraint (Li et al. 2017)

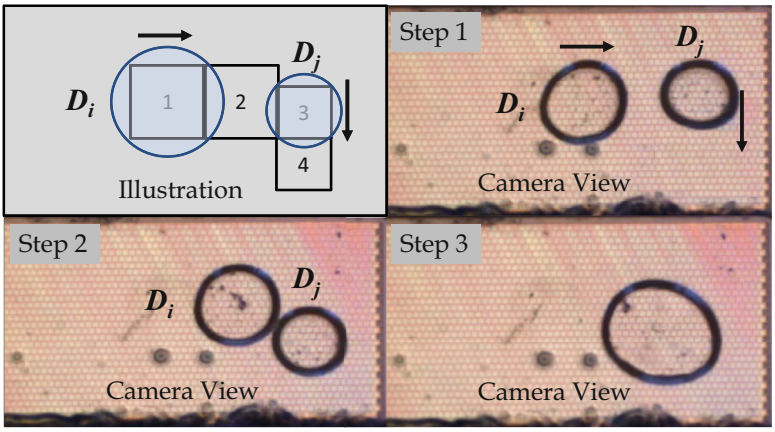


Fig. 22 Experimental verification of the dynamic-constraint (Li et al. 2017)

Chen et al. (2011) also incorporated other MEDA-specific characteristics, e.g., diagonal movements and channel-based movements. The work in Howladar et al. (2016) proposed a MEDA-based cross-reference driving scheme and routing algorithm that allow simultaneous driving of multiple droplets. The objectives of these methods include reducing the crossovers with intelligent collision avoidance, minimizing the overall routing time, and minimization of the control pin count. The work in Li et al. (2017) also proposed a modified Lee-based routing algorithm specific for MEDA biochips. A typical Lee algorithm includes four steps: initialization, wave propagation, backtrace, and clearance. Initialization aims to create routing grids and identify the source and the sink. The next step, wave propagation, progressively fills the adjacent grids with marks based on the distance of the wave front from the source to the sink. Based on the tracing information, backtrace determines the shortest path from the sink to the source. The last step, clearance, deletes all marks and preserves

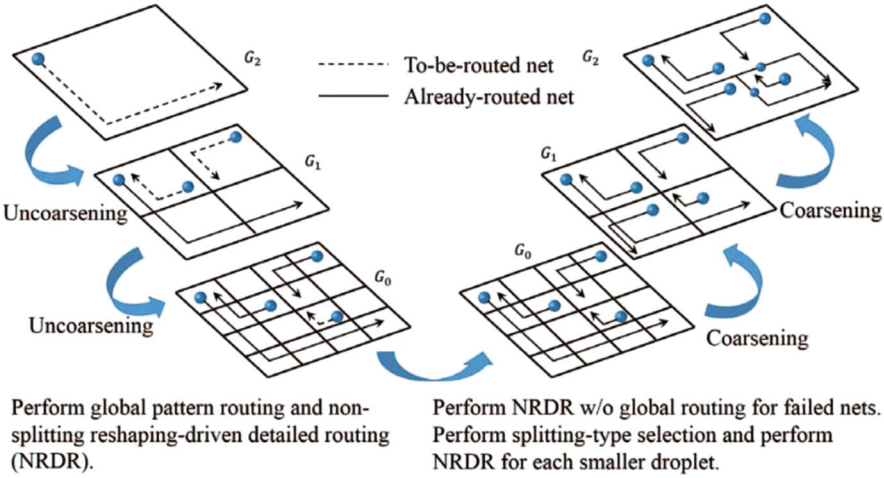


Fig. 23 An overview of the proposed multi-level droplet-routing algorithm in Lu et al. (2018)

the shortest path. In Li et al. (2017), the Lee algorithm is adopted to consider the size and shape of different droplets on a MEDA biochip. In order to incorporate the diagonal droplet transportation (Wang et al. 2011), the distance "wave" can also be propagated diagonally. Note that the droplet shape can be modified during droplet transportation (Wang et al. 2011). Droplet shape morphing during droplet routing is used for avoiding conflict with other droplets or executing fluidic modules; the droplet shape can be restored when there is no conflict. Therefore, droplet shapes as well as the relative distance should be recorded in the step of wave propagation. After the wave propagation is completed, the authors can back trace the shortest droplet route from the sink to the source.

The work in Lu et al. (2018) proposed a multi-level hierarchical approach that can take appropriate decisions on droplet splitting and reshaping. Figure 23 shows the overview of the proposed algorithm, which includes a top-down uncoarsening followed by a bottom-up coarsening. In the uncoarsening stage, a non-splitting reshaping-driven detailed routing (NRDR) algorithm was proposed to reshape droplets during droplet transportation. The reshaping decisions are guided by a proposed global droplet router. In the coarsening stage, an algorithm based on bipartite matching was proposed to select the best splitting type to split the droplets that are filed in the uncoarsening stage.

The work in Keszocze et al. (2017) proposed the first exact droplet routing technique that is capable of handling various MEDA-specific routing characteristics, such as droplet morphing and diagonal droplet transportation. At the same time, the optimal routing results can be guaranteed. The proposed method transformed the considered routing problem into a sequence of decision problems. Each decision problem is then symbolically formulated as a SAT Module Theories (SMT) instance which, afterward, is passed to a SAT solver. Satisfiability (SAT) is a fundamental

Table 6 Comparison among different routing methods

Module placement methods		Core algorithm	Feature	Runtime
Conventional DMFBs	Cho and Pan (2008)	Bypassibility & Concession based	Better routability	Medium
	Yuh et al. (2008)	Network-flow based	Concurrent routing	Fast
	Chen et al. (2013)	3D-DDM	Reliability-Oriented	Fast
	Liang et al. (2020)	Deep RL	Reliability-Oriented	Medium
MEDA Biochips	Chen et al. (2011)	A*-based	Include diagonal movements & channel-based movements	Medium
	Howladar et al. (2016)	Cross-reference- grid based	Cross-contamination-aware	Medium
	Li et al. (2017)	Lee based	Size-aware droplet router	Medium
	Lu et al. (2018)	Multi-level hierarchical	Support droplet reshaping	Medium
	Keszocze et al. (2017)	SAT based	Handling MEDA-specific characteristics	Slow

problem in mathematical logic, inference, and automated reasoning. By solving this problem through a SAT solver, a global optimum can be obtained. The resulting methodology from Keszocze et al. (2017) can be utilized to evaluate existing routing methods, such as Li et al. (2017) and Chen et al. (2011).

Qualitative comparisons among different droplet routing methods are presented in Table 6.

Conclusion

The authors have presented a survey of research on architectures and design tools for mainstream microfluidic biochips. The authors provided an overview of flow-based microfluidics, conventional digital microfluidics, and MEDA biochips. The authors also highlighted emerging applications on these biochips. Advances in simulation, synthesis, and physical design techniques have been described. These design techniques have paved the way for the deployment and use of biochips in the emerging marketplace. There is still a need for continued design

automation research for emerging biochip challenges, such as cross-contamination between samples, synthesis, and optimization under in-system error detection and automatically error recovery.

References

10xgenomics (2020). https://www.10xgenomics.com/, last accessed: August 11, 2020

Araci IE, Quake SR (2012) Microfluidic very large scale integration (mVLSI) with integrated micromechanical valves. Lab Chip 12(16):2803–2806

Chakrabarty K, Fair RB, Zeng J (2010) Design tools for digital microfluidic biochips: toward functional diversification and more than moore. IEEE Trans Comput-Aided Des Integr Circuits Syst 29(7):1001–1017

Chen Z, Teng DH-Y, Wang GC-J, Fan S-K (2011) Droplet routing in high-level synthesis of configurable digital microfluidic biochips based on microelectrode dot array architecture. BioChip J 5(4):343–352

Chen Y-H, Hsu C-L, Tsai L-C, Huang T-W, Ho T-Y (2013) A reliability-oriented placement algorithm for reconfigurable digital microfluidic biochips using 3-D deferred decision making technique. IEEE Trans Comput-Aided Des Integr Circuits Syst 32(8):1151–1162

Cho M, Pan DZ (2008) A high-performance droplet routing algorithm for digital microfluidic biochips. IEEE Trans Comput-Aided Des Integr Circuits Syst (TCAD) 27(10):1714–1724

Chung W-C, Cheng P-Y, Li Z, Ho T-Y (2018) Module placement under completion time uncertainty in micro-electrode-dot-array digital microfluidic biochips. IEEE Trans Multi-Scale Comput Syst 4(4):811–821

Crites B, Kong K, Brisk P, Diagonal component expansion for flow-layer placement of flow-based microfluidic biochips. ACM Trans Emb Comput Syst 16(5s):1–18

FDA advisors approve of Baebies SEEKER analyzer for newborns (2016). Available at https://www.baebies.com/fda-advisors-back-approval-baebies-seeker-analyzer-newborns/.

FDA advisors back approval of Baebies' SEEKER analyzer for newborns (2020). http://baebies.com/fda-advisors-back-approval-baebies-seeker-analyzer-newborns, last accessed: August 8, 2020

FDA advisors approve of Baebies SEEKER 1.5 for SARS-CoV-2 test (2021). Available at https://baebies.com/products/sars-cov-2-rt-pcr-test/

Fidalgo LM, Maerkl SJ (2011) A software-programmable microfluidic device for automated biology. Lab Chip 11(9):1612–1619

Fluidigm (2020). https://www.fluidigm.com/, last accessed: August 8, 2020

Genmark dx (2020). https://www.genmarkdx.com/, last accessed: August 8, 2020

Grimmer A, Wang Q, Yao H, Ho T-Y, Wille R (2017) Close-to-optimal placement and routing for continuous-flow microfluidic biochips. In: Proceedings of Asia and South Pacific Design Automation Conference, pp 530–535

Hengxin bio (2020). http://www.hengxinbio.com/en/index.aspx, last accessed: August 8, 2020

Howladar P, Roy D, Roy P, Rahaman H (2016) Cross-reference EWOD driving scheme and cross-contamination aware net placement technique for MEDA based DMFBs. In: Proceedings of IEEE International Conference on Advances in Computing, Communications and Informatics (ICACCI), pp 614–619

Huang T-W, Lin C-H, Ho T-Y (2009) A contamination aware droplet routing algorithm for digital microfluidic biochips. In: 2009 IEEE/ACM International Conference on Computer-Aided Design-Digest of Technical Papers. IEEE, pp 151–156

Huang X, Ho T-Y, Guo W, Li B, Schlichtmann U (2019a) MiniControl: synthesis of continuous-flow microfluidics with strictly constrained control ports. In: Proceedings of Design Automation Conference, vol 145, pp 1–6

Huang X, Ho T-Y, Chakrabarty K, Guo W (2019b) Timing-driven flow-channel network construction for continuous-flow microfluidic biochips. IEEE Trans Comput-Aided Des. Integr Circuits Syst 39(6):1314–1327

Huang S, Connolly J, Khlystov A, Fair RB (2020) Digital microfluidics for the detection of selected inorganic ions in aerosols. Sensors 20(5):1281

Huang X, Guo W, Chen Z, Li B, Ho T-Y, Schlichtmann U (2021a) Flow-based microfluidic biochips with distributed channel storage: synthesis, physical design, and wash optimization. IEEE Trans Comput 71(2):464–478

Huang X, Pan Y, Zhang GL, Li B, Guo W, Ho T-Y, Schlichtmann U (2021b) Pathdriver+: enhanced path-driven architecture design for flow-based microfluidic biochips. IEEE Trans Comput-Aided Des Integr Circuits Syst 41(7):2185–2198

Huang X, Pan Y, Chen Z, Guo W, Wang L, Li Q, Wille R, Ho T-Y, Schlichtmann U, Design automation for continuous-flow lab-on-a-chip systems: a one-pass paradigm. IEEE Trans Comput-Aided Des Integr Circuits Syst 42(1):327–331

Hu K, Dinh TA, Ho T-Y, Chakrabarty K (2014) Control-layer optimization for flow-based mVLSI microfluidic biochips. In: Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems, pp 1–10

Ibrahim M, Chakrabarty K, Schlichtmann U (2018a) Synthesis of a cyberphysical hybrid microfluidic platform for single-cell analysis. IEEE Trans Comput-Aided Des Integr Circuits Syst 38(7):1237–1250

Ibrahim M, Sridhar A, Chakrabarty K, Schlichtmann U (2018b) Synthesis of reconfigurable flow-based biochips for scalable single-cell screening. IEEE Trans Comput-Aided Design Integr Circuits Syst 38(12):2255–2270

Keszocze O, Li Z, Grimmer A, Wille R, Chakrabarty K, Drechsler R (2017) Exact routing for micro-electrode-dot-array digital microfluidic biochips. In: Proceedings of Asia and South Pacific Design Automation Conference (ASP-DAC), pp 708–713

Lai KY-T, Shiu M-F, Lu Y-W, Ho Y, Kao Y-C, Yang Y-T, Wang G, Liu K-M, Chang H-C, Lee C-Y (2015a) A field-programmable lab-on-a-chip with built-in self-test circuit and low-power sensor-fusion solution in 0.35 μm standard cmos process. In: Proceedings of the IEEE Asian Solid-State Circuits Conference (A-SSCC), pp 1–4

Lai KY-T, Yang Y-T, Lee C-Y (2015b) An intelligent digital microfluidic processor for biomedical detection. J Signal Process Syst 78(1):85–93

Lai G-R, Lin C-Y, Ho T-Y (2018), Pump-aware flow routing algorithm for programmable microfluidic devices. In: Proceedings of Design, Automation, and Test Europe Conference, pp 1405–1410

Lamanna J, Scott EY, Edwards HS, Chamberlain MD, Dryden MD, Peng J, Mair B, Lee A, Chan C, Sklavounos AA et al (2020) Digital microfluidic isolation of single cells for-omics. Nat Commun 11(1):1–13

Liang T-C, Zhong Z, Bigdeli Y, Ho T-Y, Chakrabarty K, Fair R (2020) Adaptive droplet routing in digital microfluidic biochips using deep reinforcement learning. In: Proceedings of International Conference on Machine Learning

Li M, Tseng T-M, Li B, Ho T-Y, Schlichtmann U (2016) Sieve-valve-aware synthesis of flow-based microfluidic biochips considering specific biological execution limitations. In: Proceedings of Design, Automation, and Test Europe Conference, pp 624–629

Li Z, Lai KY-T, Yu P-H, Chakrabarty K, Ho T-Y, Lee C-Y (2017) Droplet size-aware high-level synthesis for micro-electrode-dot-array digital microfluidic biochips. IEEE Trans Biomed Circuits Syst (TBioCAS) 11(3):612–626

Li Z, Lai KY-T, McCrone J, Yu P-H, Chakrabarty K, Pajic M, Ho T-Y, Lee C-Y (2018) Efficient and adaptive error recovery in a micro-electrode-dot-array digital microfluidic biochip. IEEE Trans Comput-Aided Des Integr Circuits Syst 37(3):601–614

Lin C-X, Liu C-H, Chen I-C, Lee D, Ho T-Y (2014) An efficient bi-criteria flow channel routing algorithm for flow-based microfluidic biochips. In: Proceedings of the Design Automation Conference, pp 1–6

Liu C, Huang X, Li B, Yao H, Pop P, Ho T-Y, Schlichtmann U (2021) DCSA: distributed channel-storage architecture for flow-based microfluidic biochips. IEEE Trans Comput-Aided Des Integr Circuits Syst 40(1):115–128

Lu G-R, Bhattacharya BB, Ho T-Y, Chen H-M (2018) Multi-level droplet routing in active-matrix based digital-microfluidic biochips, In: Proceedings of Asia and South Pacific Design Automation Conference (ASP-DAC), pp 46–51

Luo Y, Chakrabarty K, Ho T-Y (2012) Dictionary-based error recovery in cyberphysical digital-microfluidic biochips. In: IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2012, pp 369–376

Luo Y, Bhattacharya BB, Ho T-Y, Chakrabarty K, Design and optimization of a cyberphysical digital-microfluidic biochip for the polymerase chain reaction. IEEE Trans Comput-Aided Design Integr Circuits Syst 34(1):29–42

Minhass WH, Pop P, Madsen J, Ho T-Y (2013) Control synthesis for the flow-based microfluidic large-scale integration biochips. In: Proceedings Asia and South Pacific Design Automation Conference, pp 205–212

Najjar D, Rainbow J, Sharma Timilsina S, Jolly P, De Puig H, Yafia M, Durr N, Sallum H, Alter G, Li JZ et al (2022) A lab-on-a-chip for the concurrent electrochemical detection of SARS-CoV-2 RNA and anti-SARS-CoV-2 antibodies in saliva and plasma. Nat Biomed Eng 6(8):968–978

NOWDiagnostics (2020). https://nowdx.com/, last accessed: August 8, 2020

O'neal K, Grissom D, Brisk P (2017) Resource-constrained scheduling for digital microfluidic biochips. ACM J Emerg Technol Comput Syst (JETC) 14(1):7

Perkel JM (2008) Life science technologies: microfluidics bringing new things to life science. Science 322(5903):975–977

Ricketts AJ, Irick K, Vijaykrishnan N, Irwin MJ (2006) Priority scheduling in digital microfluidics-based biochips. In Proceedings of the Conference on Design, Automation and Test in Europe (DATE), pp 329–334

Schneider A, Pop P, Madsen J (2018) Pin-count reduction for continuous flow microfluidic biochips. Microsyst Technol 24(1):483–494

Sista RS, Ng R, Nuffer M, Basmajian M, Coyne J, Elderbroom J, Hull D, Kay K, Krishnamurthy M, Roberts C et al (2020) Digital microfluidic platform to maximize diagnostic tests with low sample volumes from newborns and pediatric patients. Diagnostics 10(1):21

Su F, Chakrabarty K (2005) Unified high-level synthesis and module placement for defect-tolerant microfluidic biochips. In: Proceedings of the Design Automation Conference (DAC), pp 825–830

Su F, Chakrabarty K (2006) Module placement for fault-tolerant microfluidics-based biochips. ACM Trans Des Autom Electr Syst (TODAES) 11(3):682–710

Su F, Chakrabarty K (2008) High-level synthesis of digital microfluidic biochips. ACM J Emerg Technol Comput Syst (JETC) 3(4):1–32

Su F, Hwang W, Chakrabarty K, Droplet routing in the synthesis of digital microfluidic biochips. In: Proceedings of DATE, vol 1. pp 1–6

Tseng K-H, You S-C, Liou J-Y, Ho T-Y (2013) A top-down synthesis methodology for flow-based microfluidic biochips considering valve-switching minimization. In: Proceedings of the International Symposium on Physical Design, pp 123–129

Tseng T-M, Li B, Schlichtmann U, Ho T-Y (2015) Storage and caching: synthesis of flow-based microfluidic biochips. IEEE Des Test 32(6):69–75

Tseng T-M, Li B, Li M, Ho T-Y, Schlichtmann U (2016) Reliability-aware synthesis with dynamic device mapping and fluid routing for flow-based microfluidic biochips. IEEE Trans Comput-Aided Design Integr Circuits Syst 35(12):1981–1994

Tseng T-M, Li M, Freitas DN, McAuley T, Li B, Ho T-Y, Araci IE, Schlichtmann U, Columba 2.0: a co-layout synthesis tool for continuous-flow microfluidic biochips. IEEE Trans Comput-Aided Des Integr Circuits Syst 37(8):1588–1601

Tseng T-M, Li M, Zhang Y, Ho T-Y, Schlichtmann U, Cloud Columba: accessible design automation platform for production and inspiration. In: Proceedings of International Conference Computer-Aided Design, pp 1–6

Urbanski JP, Thies W, Rhodes C, Amarasinghe S, Thorsen T, Digital microfluidics using soft lithography. Lab Chip 6(1):96–104

Urbanski J, Thies B, Amarasinghe S, Thorsen T (2007) (Online) Programmable microfluidics. [Online]. Available: http://groups.csail.mit.edu/cag/biostream/

Wang G, Teng D, Fan S-K (2011) Digital microfluidic operations on micro-electrode dot array architecture. IET Nanobiotechnol 5(4):152–160

Wang Q, Ru Y, Yao H, Ho T-Y, Cai Y (2016) Sequence-pair-based placement and routing for flow-based microfluidic biochips. In: Proceedings of Asia and South Pacific Design Automation Conference, pp 587–592

Wang Q, Xu Y, Zuo S, Yao H, Ho T-Y, Li B, Schlichtmann U, Cai Y (2017) Pressure-aware control layer optimization for flow-based microfluidic biochips. IEEE Trans Biomed Circuits Syst 11(6):1488–1499

Wu J-L, Li KS-M, Li J-D, Wang S-J, Ho T-Y (2018) SOLAR: simultaneous optimization of control-layer pins placement and channel routing in flow-based microfluidic biochips. In: Proceedings of International Symposium VLSI Design Automation Test, pp 1–4

Xu T, Chakrabarty K (2007) Integrated droplet routing in the synthesis of microfluidic biochips. In: Proceedings of Design Automation Conference, pp 948–953

Yang K, Yao H, Ho T-Y, Xin K, Cai Y (2018) AARF: any-angle routing for flow-based microfluidic biochips. IEEE Trans Comput-Aided Des Integr Circuits Syst 37(12):3042–3055

Yao H, Wang Q, Ru Y, Cai Y, Ho T-Y (2015) Integrated flow-control codesign methodology for flow-based microfluidic biochips. IEEE Des Test 32(6):60–68

Yao H, Ho T-Y, Cai Y (2015) PACOR: practical control-layer routing flow with length-matching constraint for flow-based microfluidic biochips. In: Proceedings of Design Automation Conference, pp 1–6

Yuh P-H, Yang C-L, Chang Y-W (2008) Bioroute: a network-flow-based routing algorithm for the synthesis of digital microfluidic biochips. IEEE Trans Comput-Aided Des Integr Circuits Syst (TCAD) 27(11):1928–1941

Zhu Y, Huang X, Li B, Ho T-Y, Wang Q, Yao H, Wille R, Schlichtmann U, MultiControl: advanced control logic synthesis for flow-based microfluidic biochips. IEEE Trans Comput-Aided Des Integr Circuits Syst 39(10):2489–2502

Architectures for Quantum Information Processing

21

Suryansh Upadhyay, Mahabubul Alam, and Swaroop Ghosh

Contents

Introduction . . . 724
Background . . . 725
Quantum Bits (Qubits) . . . 726
Quantum Gates . . . 727
Quantum Error . . . 727
Quantum Hardware . . . 729
Qubit Technologies . . . 730
Quantum Algorithms . . . 732
Algorithms Designed for Fault-Tolerant Quantum Computers . . . 733
Algorithms for NISQ Computers . . . 734
Quantum Software . . . 735
Quantum Program, Quantum Instruction Sets, and Software Development Kits . . . 735
Quantum Programming Languages . . . 736
Quantum Annealing . . . 738
Compilation, Mapping, and Optimization . . . 739
Superconducting Quantum Computers . . . 740
Trapped-Ion Quantum Computers . . . 741
Considerations for Noisy Systems . . . 743
Technology Agnostic Work . . . 743
Superconducting-Specific Work . . . 744
Conclusion . . . 746
References . . . 746

S. Upadhyay · M. Alam · S. Ghosh (✉)
Pennsylvania State University, University Park, PA, USA
e-mail: sju5079@psu.edu; mxa890@psu.edu; szg212@psu.edu

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_64

Abstract

Quantum computing is changing the way people think about computing. Significant strides in research and development for managing and harnessing the power of quantum systems has been made in recent years, demonstrating the potential for transformative quantum technology. Quantum phenomena like superposition, entanglement, and interference can be exploited to solve issues that are difficult for traditional computers. IBM's first public access to true quantum computers through the cloud and Google's demonstration of quantum supremacy are among the accomplishments. Besides, a slew of other commercial, government, and academic projects are in the works to create next-generation hardware, a software stack to support the hardware ecosystem, and viable quantum algorithms. This chapter covers various quantum computing architectures including many hardware technologies that are being investigated. It also discusses a variety of challenges, including numerous errors/noises that plague the quantum computers. An overview of literature investigating noise-resilience approaches is also presented.

Keywords

Quantum computing · Quantum software · Quantum algorithms · Quantum hardware

Introduction

Quantum computing (QC) exploits phenomena such as superposition, entanglement, and interference to efficiently explore exponentially large state spaces and compute solutions for certain classically intractable problems. Despite the extraordinary development in the classical computing domain, including device, architecture, and algorithm in the past decades, many problems cannot be solved in a reasonable time, even on the fastest classical supercomputer. For example, solving the nitrogen fixation problem on the best classical supercomputer would take more than 2 million years (Reiher et al. 2017). Since 1982, when Richard Feynman originally envisioned the quantum-mechanical computer, researchers have taken incremental but significant efforts toward developing quantum algorithms and hardware. The original Church-Turing thesis in its contrapositive form states that a computation that cannot be performed by a Turing machine cannot be performed without breaking a physical law. It is a physics principle because it implies a limitation on what physical systems can do. However, quantum machines are able to perform calculations in polynomial time that Turing machines are believed to not be able to do, e.g., factorization. David Deutsch and Richard Jozsa presented the "Deutsch-Jozsa algorithm" (Deutsch and Jozsa 1907), which illustrated how a quantum algorithm might perform a job in fewer steps than a conventional version. Shor's algorithm (Shor 1999), which is

one of the most well-known quantum algorithms, can factor an integer in prime numbers tenfold quicker than the best conventional solution. The impact of this exponential speedup on encryption and Internet security is substantial. Cirac and Zoller suggested the experimental implementation of the Controlled NOT (CNOT) gate (Cirac and Zoller 1995) on the hardware side. Following that, nuclear magnetic resonance (NMR)-based devices were used to demonstrate quantum computing hardware, including the first demonstration of a full-fledged quantum algorithm on a 2-qubit NMR computer at Oxford University. Schoelkopf, Devoret, Girvin, and colleagues at Yale University invented the superconducting "Transmon" qubit, which revolutionized qubit technology and paved the road for scalability. IBM first provided access to a 5-qubit programmable quantum computer through the cloud. The free access to quantum computers attracted many researchers around the globe to the world of quantum computing. The quantum threshold theorem (Shor 1994) demonstrated that if the error to perform each gate is a small enough, one can perform arbitrarily long quantum computations to arbitrarily good precision with only a small increase in gate count. This shows that quantum computers can be made fault-tolerant. Especially in the last few years, quantum computing has experienced breakthroughs across the stack, including algorithm, architecture, and hardware. The most notable of these is the demonstration of Quantum Supremacy by Google (Arute et al. 2019). The group of researchers from Google performed a task on a 53-qubit quantum computer in seconds which arguably would take days on the fastest classical supercomputer. Application domains of QC now include machine learning, security, drug discovery, computational quantum chemistry, and optimization.

On the one hand, researchers are proposing new quantum algorithms to speed up computation, while on the other hand, various technologies like superconducting, trapped-ion (TI), and photonics are also being studied to design efficient quantum bits or qubits. Despite all the signs of progress, quantum computers are yet to solve practical-scale problems.

The quantum computer architecture is essential for the functioning of a quantum computer. Despite various system optimizations, the performance of a quantum processor is still severely constrained by the amount of available computation resource. This chapter covers the fundamentals of quantum computing architectures including some of the most often used hardware architectures and the associated software stacks. It also discusses the numerous issues that these technologies face, as well as a literature review of efforts to address these architectural issues for both hardware and software stacks. Figure 1 illustrates the topics covered in this chapter.

Background

This section gives a quick introduction of quantum computing's fundamental ideas in order to provide readers an understanding of the new computing paradigm allowed by quantum properties and the associated challenges.

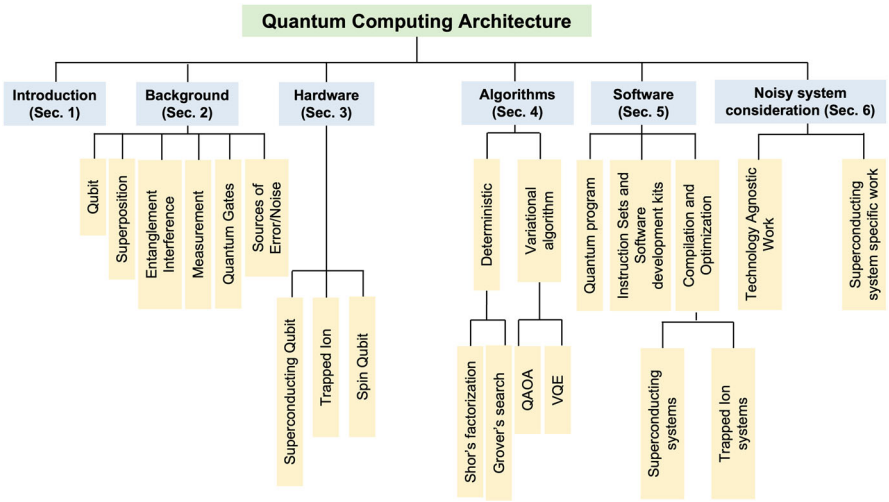


Fig. 1 Chapter structure

Quantum Bits (Qubits)

In quantum computing, a qubit is the basic unit of quantum information, the quantum version of the classic binary bit physically realized with a two-state device. For example, electron spin can realize a qubit where electron up-spin can represent data "1" and down-spin can represent data "0." Therefore, a qubit has two quantum states, analogous to the classical binary states. However, classical bit can be either "0" or "1" at a time, and qubits can be a mixture of both 0 and 1 simultaneously due to quantum superposition. Mathematically, a qubit state is represented by state vector $\psi = a\,|0\rangle + b\,|1\rangle$ where $|a|^2$ and $|a|^2$ represent probabilities of "0" and "1," respectively. Suppose a qubit is in state $\psi = .707\,|0\rangle + .707\,|1\rangle$. Reading out this multiple times will theoretically generate 0s and 1s with equal probability (as 0.7072 = 0.5). The state of a qubit is often represented visually by a Bloch sphere (Fig. 2), where the poles of the sphere represent states 0 and 1, and the equator is the perfect superposition. Furthermore, qubit states can be entangled, allowing two or more qubit states to be correlated. The states of other entangled qubits can be changed by performing a single operation on one of the entangled qubits. Quantum interference is a property of qubit states that algorithm designers can take advantage of. A qubit state's amplitude can be both positive and negative. As a result, a quantum algorithm designer can tweak the gate operations so that a negative amplitude of the same qubit state cancels out a positive amplitude. Quantum superposition, entanglement, and interference, according to researchers, are at the heart of quantum speed-ups of quantum algorithms. Upon measurement, the qubit's coefficient (or amplitude) becomes one in the state that is read and zero in the other; all information about the amplitudes is destroyed upon measurement also known as collapse of state.

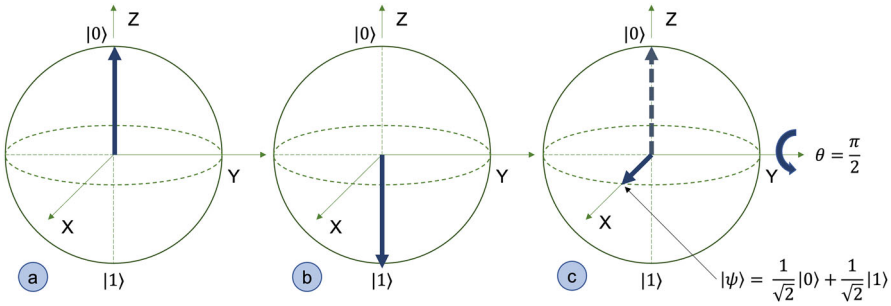


Fig. 2 Bloch sphere representation of state (**a**) $|0\rangle$ and state (**b**) $|1\rangle$. (**c**) Bloch sphere representation of the $R_Y(\pi/2)$ gate on state $|0\rangle$

Quantum Gates

Gates are used in quantum computing systems to regulate qubit amplitudes and execute computations. At any given time, gates can act on one or more qubits. QC systems often support a set of universal single-qubit and two-qubit gates, similar to universal gates in classical computing. Quantum gates, unlike classical logic gates, are not physically formed; instead, they are realized through the use of pulses. These gate sets are used to express QC applications. A sequence of gates is executed on a set of correctly initialized qubits to run a program. The gates change the amplitudes of the qubits, moving the state space closer to the desired output. Intuitively, the gate pulses cause distinct rotations along different axes in the Bloch sphere (depending on pulse amplitude, duration, and shape). For example, the $R_Y(\pi/2)$ (rotation along Y-axis) can be a quantum gate, and it will rotate a qubit state by $\pi/2$ radian around Y-axis (e.g., applying an $R_Y(\pi/2)$ will put a qubit in $|0\rangle$ state in the superposition state, Fig. 2c).

Mathematically, quantum gates are represented using unitary matrices (a matrix U is unitary if $UU^{\dagger} = I$, where $U^{\dagger}$ is the adjoint of matrix U and I is the identity matrix). For an n-qubit gate, the dimension of the unitary matrix is $2n \times 2n$. Any unitary matrix can be a quantum gate. However, in existing systems, only a handful of gates are possible, often known as the native gates or basis gates of that quantum processor. For IBM systems, the basis gates are ID, RZ, SX, X, and CNOT. CNOT is the only 2-qubit gate, and others are single-qubit. Any non-native gate in a quantum circuit is first decomposed using the native gates.

Quantum Error

Quantum systems are plagued with noise because of the quantum gates being error-prone. Besides, the qubits suffer from decoherence, i.e., the qubits spontaneously interact with the environment and lose states. Therefore, the output of a quantum circuit is erroneous. The deeper quantum circuit needs more time to execute and gets

affected by decoherence. More gates in the circuit also increase the accumulation of gate error. Parallel gate operations on different qubits can affect each other's performance which is known as crosstalk. This section elaborates on these errors:

Gate Error

Quantum gates are realized with pulses, and the pulses can be erroneous. For example, consider the $R_Y(\pi/2)$ gate. Due to variation, the pulse intended for a $\pi/2$ rotation may not result in an exact $\pi/2$ rotation, and it may under-rotate or over-rotate, leading to erroneous logical operation. As a result, gate failures are caused by faulty logical operations. For present systems, 2-qubit gate errors (e.g., CNOT error) are an order of magnitude larger than 1-qubit gate faults. A quantum circuit with a larger number of gates will accrue more gate faults, lowering the quantum program's reliability. Hence, compilation and error-tolerant mapping have the goal of reducing the number of gates in the quantum circuit.

Relaxation and Dephasing

Decoherence is related to a short qubit lifetime. Qubits may interact with the environment and spontaneously lose their saved state. For example, Fig. 3 shows the effect of relaxation, one type of decoherence. Due to relaxation, a qubit in state $|1\rangle$ spontaneously loses energy and ends up in state 0. Decoherence of a qubit is usually characterized by T1 relaxation time and T2 dephasing time. If the gate time is tg, then roughly 1–exp(–tg/T1) is the error probability that a state $|1\rangle$ will be damped. This implies that if the gate time (operation) is long, the qubit will lose its state more.

Measurement Error

Reading out a qubit containing a state 1 may result in a state 0 and vice versa due to readout error; this arises due to measuring circuitry imperfections. The readout error probability can be quantified using a simple technique. It entails preparing a qubit in all binary states (i.e., 0 and 1 for a single qubit) and reading it out (both preparation

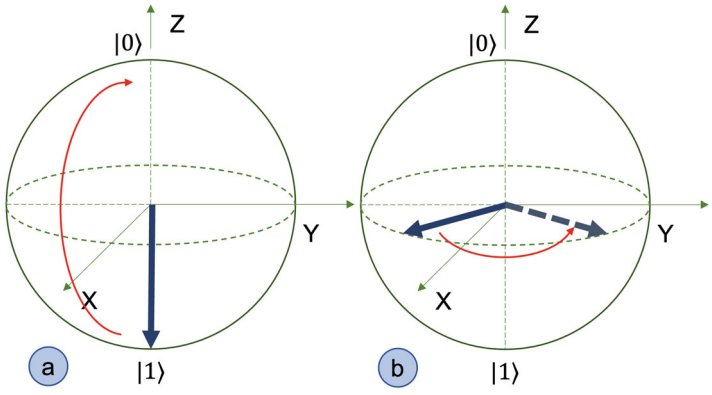


Fig. 3 Illustration of (**a**) relaxation and (**b**) dephasing of qubit states

and measurement multiple times). The qubits on IBM machines are initially set to 0 states by default. Therefore, to prepare state "1," a quantum-NOT (X) gate has to be applied to the $|0\rangle$ state. Ideally, if the process of preparing a state 0 or 1 and reading out N times is repeated, it should generate 0 or 1 all the time. However, due to readout error, a flipped bit might be read in some of the cases. For example, say state 0 is prepared and measured 1000 times. Out of these 1000 trials, 900 times it reads out 0, and other 100 times it reads out 1. Thus, measurement error rate M_{01} will be 100/1000 = 0.1 (M_{xy} stands for probability of reading out state "y," while the prepared state is "x"; thus, $M_{00} = 900/1000 = 0.90$). For multi-qubit readout characterization, the same protocol applies. However, the number of binary states that need to be prepared and read will be higher. For example, to characterize a 3-qubit system, $2^3 = 8$ binary states (000, 001, 010, …, 110, and 111) need to be prepared and measured (each N-times). Unlike gate error and decoherence, which depend on the number of gates in the circuit, readout error is gate count agnostic. It solely depends on the state being read.

Crosstalk Error

Crosstalk is another kind of error present in the near-term quantum computers. The effect of a gate operation on one qubit should, in theory, be unaffected by what happens on other qubits. Pulses are used to create quantum gates. However, the gate pulse intended for one qubit can accidentally excite an unwanted qubit, which is known as "crosstalk." Crosstalk may cause conditional dependence in gate errors. As a result, the gate error of a single gate operation may differ from the gate error of a parallel gate operation. According to Murali et al. (2020a), the gate error with another operation running in parallel can be 2X–3X higher than with an isolated gate operation.

Quantum Hardware

Having introduced quantum computing basics in the prior section, this section focuses on the various hardware technologies and developments (Fig. 4).

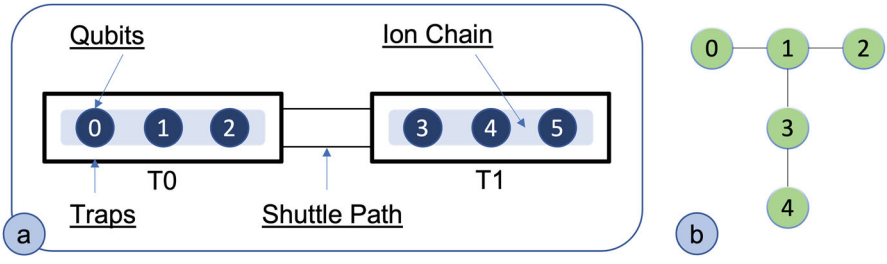


Fig. 4 (**a**) A two-trap TI system with three qubits each. (**b**) Coupling graph for ibmq_lima superconducting device

Qubit Technologies

A qubit is the fundamental unit of quantum information and the foundation of a quantum computer. At their heart, qubits are two-level systems, which means that every two-level system can physically realize a qubit. There are several technologies such as superconducting, trapped-ion, neutral atom, diamond NV-center, quantum dot, and photon that satisfy the requirement for a qubit. Some of the most common types are:

Superconducting Qubits

Superconductors allow an electrical current to flow with no resistance when cooled to very low temperatures. Electrical circuits based on superconductors that behave like qubits can be designed. The idea is to build an anharmonic oscillator. In an anharmonic oscillator, the energy separation between states is different. Therefore, the lowest two energy states are used as a qubit. For harmonic oscillators, the energy states are equally separated, which makes it difficult to control interstate transition. Superconducting qubits are fabricated by connecting a capacitor and a superconducting Josephson junction (JJ) in parallel. This assembly works as an anharmonic LC oscillator in which the Josephson junction works as a nonlinear inductor. The JJ requires ultralow temperature for it to operate in the superconducting regime. Thus, superconducting qubits are usually hosted inside large dilution refrigerators. Kjaergaard et al. (2020) gives a comprehensive overview of the current state of play for superconducting qubits. Prominent companies conducting research in superconducting quantum computing are Google, Rigetti, IMEC, BBN Technologies, Intel, and IBM. According to Li et al. (2021), quantum software and hardware systems should be designed collaboratively in order to fully exploit the potential of quantum computing. They review several architectural design works. One of them is developing a superconducting quantum processor architecture for a specific program in order to achieve a high yield rate with a low mapping overhead. The proposed architecture design flow is depicted in Fig. 5. They divided the design of a superconducting quantum processor architecture into three key subroutines: layout design, bus selection, and frequency allocation. Each subroutine targets a different hardware component or configuration, incorporating profiling results and

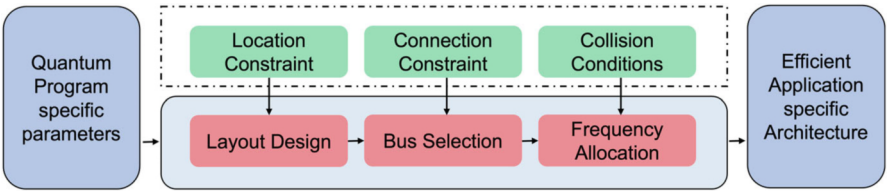


Fig. 5 Overview of quantum application-specific architecture design flow. (Adopted from Li et al. 2021)

physical constraints. They focus on the qubit placement in the layout design and try to make those qubit pairs with more two-qubit gates between them nearby to reduce the mapping overhead. The bus selection subroutine then determines how the physical qubits are linked. According to the profiling information, they only add qubit connections (also known as qubit buses) to the locations that are expected to reduce the mapping overhead the most. Finally, the frequency allocation function will assign frequencies to all physical qubits that have been put. By attempting to eliminate frequency collision scenarios on the created architecture, the subroutine will boost the final yield rate.

Trapped-Ion Qubits

Another way of realizing a qubit is by using the energy levels of electrons in neutral atoms or ions. In their natural state, these electrons occupy the lowest possible energy levels. Lasers are used to "excite" them to a higher energy level and can assign the qubit values based on their energy state. Trapped-ion QC system is implemented by trapping ionized atoms like Yb or Ca between electrodes using electromagnetic field (Wright et al. 2019). Data $|0\rangle$ and $|1\rangle$ are encoded as internal states such as hyperfine or Zeeman states of the ions. Qubits are stored in stable electronic states of each ion, and quantum information can be transferred through the collective quantized motion of the ions in a shared trap (interacting through the Coulomb force). Figure 4a, illustrates various components of a 2-trap TI system. The ions are organized in the form of an ion chain inside a trap. Trap capacity is the maximum number of ions that a trap can accommodate. The traps are connected by a shuttle path which allows movement (shuttle) of an ion from one trap to another if needed. Prominent companies conducting research in trapped-ion quantum computing are IonQ, Honeywell, Alpine Quantum Technologies, and Universal Quantum. TI systems typically employ a single trap design, which has significant scaling issues. A modular design known as the quantum charge-coupled device (QCCD) has been proposed (Murali et al. 2020b) to advance toward the next significant milestone of 50–100 qubit TI devices. Small traps are coupled by ion shuttling in a QCCD-based TI system. Authors conduct an intensive application-driven architecture analysis to evaluate the major design choices of trap size, communication topology, and operation implementation methodologies in order to realize QCCD-based TI systems with 50–100 qubits. They show that trap sizing and communication topology decisions can affect application dependability by up to three orders of magnitude using several applications as benchmarks and several hardware design points. Another approach to design a hardware architecture for TI systems is discussed in Wu et al. (2021). The authors propose adopting "TILT" (Fig. 6), a linear "Turing machine-like" architecture with a multi-laser control "head" in which a linear chain of ions moves back and forth under the laser head, as a building block to extend previous scalable trapped-ion quantum computing approaches. They claim that TILT can significantly decrease communication when compared to quantum charge-coupled device (QCCD) systems of comparable size. The principle behind a TILT design is that operations are only done to ions in the

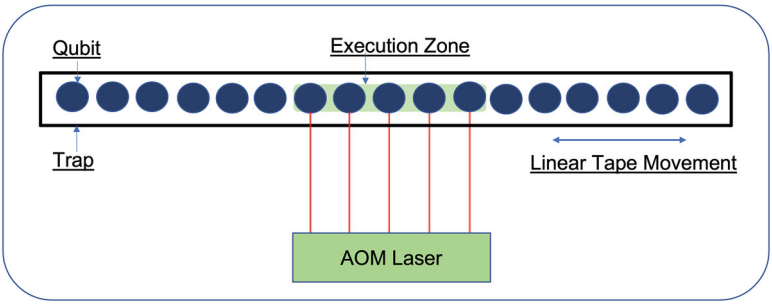


Fig. 6 A quantum computer architecture based on trapped ions and linear tapes. Acousto-optic modulators (AOMs) aim laser beams onto ions in the execution zone to perform quantum operations. The entire ion chain is translated until the target qubit is relocated into the execution zone in order to execute gate operations on the other qubits. (Adopted from Wu et al. 2021)

execution zone toward the center of the trap, and the chain is moved back and forth to allow for long-range interactions. The complex shuttling primitives of a quantum charge-coupled device (QCCD) design are hence not required for such a machine.

Spin Qubits

Controlling the spin of charge carriers (electrons and electron holes) in semiconductor devices can also be used to implement a qubit (Chatterjee et al. 2021). The majority of quantum particles act like tiny magnets. Spin is the name for this characteristic. The spin orientation is either entirely up or fully down, never halfway up or down. A spin qubit is created by combining these two states. Local depletion of two-dimensional electron vapors in semiconductors such as gallium arsenide, silicon, and germanium has been used to create spin qubits. Some reports also show implementation in graphene (Trauzettel et al. 2007).

Quantum Algorithms

Quantum algorithms are algorithms that run on a realistic model of quantum computation (the most used one being the quantum circuit model (Nielsen and Chuang 2002)) and that are inherently quantum or use some essential feature of quantum computation such as quantum superposition or quantum entanglement. Problems that are fundamentally unsolvable by classical algorithms (Lanzagorta and Uhlmann 2009) cannot be solved by quantum algorithms either. However, the added value of quantum algorithms is that they can solve some problems significantly faster than classical algorithms as they leverage quantum phenomena such as superposition, entanglement, and interference. Applications of quantum computing are as diverse as the fields necessary to create quantum information processing technology. There is an extensive literature on quantum algorithms that has been developed (Montanaro 2016). The field is now entering the era of

noisy intermediate-scale quantum (NISQ) devices (quantum computers that are sufficiently large-tens to hundreds or a few thousand qubits that they cannot be efficiently simulated by a classical computer but are not fault tolerant). Noise was only examined formally and proved to be theoretically surmountable in the early stages of quantum computation, with considerable involvement from the mathematics and computer science communities. As a result, the first wave of quantum algorithms considered that quantum devices would operate without making any noise (or otherwise fully quantum error corrected systems). Since the introduction of NISQ devices, a second wave of quantum algorithms has sprung out, taking into account noise and breakthroughs in algorithm design on traditional computers. This section aims to provide a brief insight into the literature. This section is divided into two subsections: Algorithms Designed for Fault-Tolerant Quantum Computers and Algorithms That Run on NISQ Computers.

Algorithms Designed for Fault-Tolerant Quantum Computers

The initial quantum computing algorithms were developed with an ideal quantum computer in mind, with the quantum gate model studied largely without noise. Nielsen and Chuang (2002) is the canonical reference for this wave of quantum algorithm development, and it remains a reliable reference for the theoretical basis of quantum computing and quantum information to this day. The best-known algorithms are Shor's algorithm for factoring and Grover's algorithm for searching an unstructured database or an unordered list.

Shor's Algorithm

Shor's algorithms describe two quantum algorithms for integer factoring and discrete logarithm exponentially faster than the best-known classical algorithms (Shor 1994). Because of the apparent speedup compared to classical algorithms and the implications of this speedup for known applications, it is a notable and celebrated scientific contribution to quantum computing. Shor's algorithms take advantage of both quantum parallelism and entanglement. There are two sections to the algorithm. The first portion of the algorithm converts the factoring problem into a problem of determining a function's period and can be implemented in a traditional way. The quantum speedup is determined by the second portion, which uses the quantum Fourier transform to find the period. Essentially, the paper Shor (1994) shows that the factoring problem is equivalent to the problem of finding the period in a sequence of numbers, although a sequence of numbers that is exponentially longer than the number of bits of the corresponding number to be factored. Thus, while this equivalency does not provide any help in solving the problem on a classical computer (since it would need to generate this sequence of 2n numbers for an n-bit number to factor, which would take an exponential amount of time), it is a perfect problem for a quantum computer as it can be encoded into merely n qubits and generated in a time that is polynomial in n. Once that sequence is generated, the QFT can be used to find the period. Shor's method, if implemented on a perfect

quantum computer, would allow the secret key of the most frequently used public key cryptosystem, RSA, to be computed, meaning that public key encryption might be readily broken.

Grover's Algorithm

Grover's algorithm also known as the quantum search algorithm was introduced by Lov Grover in (1996). It is used for searching an unsorted database with N entries in $O(\sqrt{N})$ time and using O(logN) storage space. Searching an unsorted database traditionally involves a linear search, which takes O(N) time. Grover's technique, on the other hand, takes $O(\sqrt{N})$ time and is the fastest quantum algorithm for doing so. Unlike other quantum algorithms, which can provide exponential speedup over their classical equivalents, it delivers a quadratic speedup. When N is big, even quadratic speedup is significant. It may also be used to calculate the mean and median of a group of values, as well as to solve the collision problem. It can also be used to tackle NP-complete problems by doing exhaustive searches across all feasible solutions. This would result in a significant speedup as compared to traditional techniques. Grover's algorithm can also be applied to speed up broad classes of algorithms. Grover's algorithm is probabilistic like all quantum algorithms, in the sense that it gives the correct answer with high probability. The probability of failure can be decreased by repeating the algorithm.

Algorithms for NISQ Computers

The near-term quantum devices have limited number of qubits. Moreover, they suffer from various types of noises (decoherence, gate errors, measurement errors, crosstalk, etc.). Due to these constraints, these machines are not yet fully capable of executing quantum algorithms requiring high orders of error correction (such as Shor's factorization or Grover's search). However, algorithms such as quantum approximate optimization algorithm (QAOA), Variational Quantum Eigensolver (VQE) promises to achieve quantum advantage with near-term machines because they are based on a variational principle that does not necessitate error correction. Most of these approaches utilize a conventional computer to perform an optimization procedure using information extracted from the quantum device, usually in an iterative fashion. These quantum optimization methods have been applied to diverse areas such as quantum machine learning.

Variational Quantum Eigensolver or VQE

The variational quantum eigensolver (or VQE) was introduced by Peruzzo et al. (2013). The basic concept which is at the heart of VQE is that the computed energy of the ground (lowest energy) state of a quantum chemical system decreases as the approximations to the solution improve, asymptotically approaching the true value from above. The input is a rough estimate at the solution, and the output is a slightly improved approximation. This output is then utilized as a guess for the next iteration, and the output grows closer to the correct solution with each cycle. The problem is

split down into a series of smaller problems that can be estimated independently in VQE, with the sum of all outputs corresponding to the approximate solution of interest. The process is repeated until a heuristic stopping criteria is met, which is usually equivalent to reaching an energy threshold.

Quantum Approximate Optimization Algorithm or QAOA

The quantum approximate optimization algorithm (QAOA) is a hybrid quantum-classical variational algorithm designed to tackle combinatorial optimization problems (Farhi et al. 2014). To optimize an objective function, it employs classical optimization of quantum operations. The algorithm is similar to the VQE algorithm in that it starts with a series of preparation and measurement experiments before being optimized by a traditional computer. When sampled, the resulting quantum state gives approximate or exact answers to the computational task.

Quantum Software

A problem is first established and then translated/optimized by software support so that it may be solved efficiently by hardware or simulators in any computing ecosystem. A software suite typically contains programming languages, compilers for mapping algorithms to machines, simulation, debugging, and optimization tools to aid in the efficient implementation of algorithms on systems. Simulation tools, in particular for quantum computers, can allow a programmer to model each quantum operation and track the quantum state that arises, as well as its progress over time. Debugging both applications and newly built hardware need this capabilities. Resource estimators and other optimization tools would allow for quick estimation of the performance and qubit resources required to run various quantum algorithms. This enables a compiler to transform the desired computation into an efficient form, minimizing the number of qubits or qubit operations required for the hardware in question. The optimization heuristics will also depend on the type of hardware the quantum program is going to be run on. Each hardware presents a unique set of challenges. This section reviews compilation, mapping, and optimization heuristics used for two of the most common quantum hardware – superconducting quantum computers and trapped-ion quantum computers.

Quantum Program, Quantum Instruction Sets, and Software Development Kits

A quantum program can be represented in the well-adopted quantum circuit model (Nielsen and Chuang 2002). Figure 7 illustrates a brief overview of a quantum program ecosystem. It can be modeled as a series of quantum gates operating on qubits to converge the output to a given solution, similar to classical computing. A quantum program is made up of variables that are logical qubits and quantum operations that can change the state of the qubits. Quantum instruction sets are used

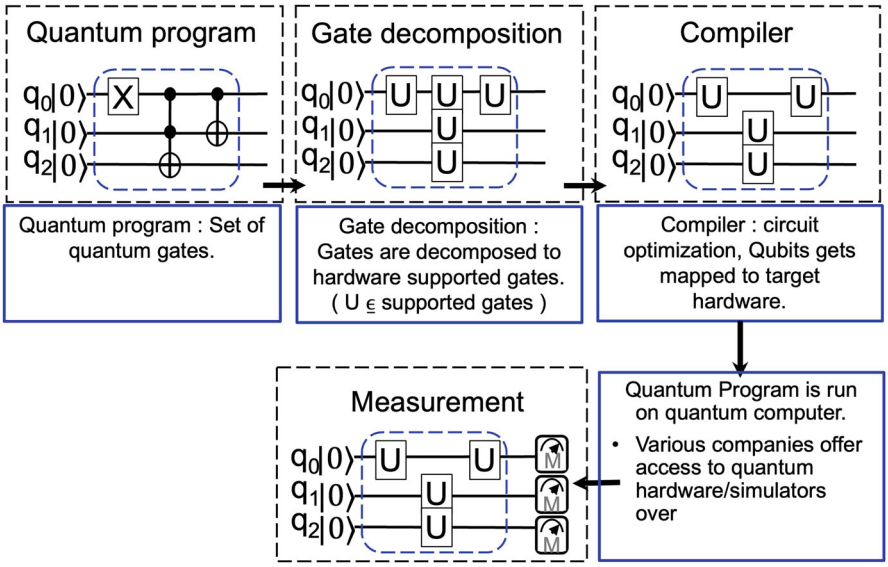


Fig. 7 Overview of program flow for quantum computers

Fig. 8 Various leading companies in quantum domain and their products

Companies	Products
IBM	IQ Experience
Google	Quantum Playground
Microsoft	LIQUi|>
Riverlane	DeltaFlow
Qutech	Quantum Inspire
QC Ware	Forge
Strangeworks	Quantum Computing Platform

to turn higher-level algorithms into physical instructions that can be executed on quantum processors. There are various instruction set architectures available such as cQASM, OpenQASM, Quil, Blackbird, etc. Quantum software development kits are bundles of tools that allow users to construct and manipulate quantum programs. The users are provided access to quantum hardware where they can simulate the quantum programs or prepare them to be run using cloud-based quantum devices. Figure 8 illustrates the companies and their products that are the leading computational platforms for quantum computing.

Quantum Programming Languages

The work published by Bettina Heim and group provides an overview of Q#, Qiskit, Cirq, Quipper, and Scaffold as well as the tools/ecosystems that surround them,

and how they have served as a foundation for current and future work (Heim et al. 2020). **Q#** is a hardware-agnostic quantum programming language designed to enable the execution of large-scale applications on future quantum hardware (Svore et al. 2018). As a result, rather than following the imperative style encouraged by assembly-like languages, Q# focuses on providing high-level abstractions that facilitate reasoning about the intended functionality. It is notable for its support for expressing arbitrary classical control flow. This is in contrast to other quantum programming languages, where this capability is frequently provided by a classical host language. Unlike other quantum programming languages geared toward formal verification, qubits in Q# are treated like any other data type. The associated libraries, the Q# compiler, and all other components of the quantum development kit are open source. **OpenQASM** is a quantum program intermediate representation based on gates (Cross et al. 2017). It expresses quantum programs as lists of instructions, which are frequently intended to be consumed directly by a quantum processor. OpenQASM supports abstractions in the form of quantum gates, which can be built in a hierarchical fashion using a set of intrinsic primitives assumed to be available on the targeted processor, for example, a Toffoli gate made up of CNOT gates, T gates, and H gates. In addition, OpenQASM supports single-qubit measurement and basic classical control operations. Qiskit provides a Python-based programming environment for creating and manipulating OpenQASM programs (Treinish et al. 2019). It includes extensive simulation capabilities, such as state vector and density matrix simulators that can be run on both CPUs and GPUs, in addition to support for execution on quantum processors. As a result, it allows users to simulate the effects of noise defined by any custom model, including arbitrary Kraus operators. The online documentation (Qiskit documentation), which includes tutorials and is generated for each release, provides a good overview of the full range of capabilities included in Qiskit. **Cirq** is a Python quantum programming library that focuses on supporting near-term quantum hardware. Cirq's primary goal is to enable the development of quantum programs capable of running on quantum computers available now or in the near future that lack error correction (NISQ hardware) and are subject to certain device topologies. It includes mechanisms for fine-tuning how a quantum program executes on the specified quantum hardware, as well as tools for simulating hardware constraints such as noise limitations or the physical layout of the qubits (Cirq documentation). In contrast to other languages where qubits can be allocated dynamically, layout in Cirq is done manually. It is built into Python. Python's control flow constructs, such as if and while test statements, can be used to build a circuit before execution. Cirq includes device models for many of Google's quantum processors (Cirq documentation), such as Bristlecone and Sycamore.

Quipper is a circuit description language, which means it can be used to construct circuits by applying gates on qubits in an organized manner. The circuits themselves are data that can be provided to functions in the host language Haskell for circuit optimization, resource estimation, or error correction, for example. Prototypical implementations of Quipper-like languages, such as Proto-Quipper-S (Ross 2015), Proto-Quipper-M (Rios and Selinger 2017), and Proto-Quipper-D

(Fu et al. 2020), have evolved with the purpose of enforcing quantum-specific features such as the quantum information no-cloning theorem. **Scaffold** is a standalone programming language. It is intended to be similar to existing traditional programming languages, like C: Scaffold uses the imperative programming model of C, as well as many of its recognizable features such as functions (called modules in Scaffold), if statements, loops, structures, and pre-processor directives (Abhari et al. 2012). Scaffold programs can also automatically convert conventional functions into reversible logic, which is done using quantum gates, and then incorporate it as an oracle in a larger quantum algorithm (Abhari et al. 2012). **Intel Quantum SDK** Intel recently demonstrated its Quantum SDK at IEEE Quantum Week, held in Colorado, USA, 2022. It provides developers with tools to help them learn how to program quantum algorithms. It is based upon the C++ programming language and uses the LLVM intermediate level description from classical computing as a base. It is designed to work with hybrid classical/quantum variational algorithms and will be compatible with other components of Intel's quantum stack, such as high-performance quantum simulators and, eventually, Intel's spin-qubit-based quantum processor. The beta version is accessible via the Intel Developer Cloud.

Other quantum programming languages and open-source software frameworks include Forest/PyQuil (Smith et al. 2016), ProjectQ (Steiger et al. 2018), QWIRE (Green et al. 2013), staq (Staq-GitHub), Strawberry Fields (Strawberry fields), tket (tket-GitHub), XACC (McCaskey et al. 2020), and QuTiP (Qutip documentation).

Quantum Annealing

There are various approaches to building quantum computing hardware, such as universal gate model quantum computers or quantum annealers. Universal gate model quantum computing, also known as general-purpose quantum computing, is the most powerful and flexible type of quantum computer, but it is difficult to build and maintain the stability of qubits. It is based on creating quantum circuits with stable qubits and using them to solve problems. However, maintaining stability for qubits is difficult. As the number of qubits increases, so does the complexity of the problem. Quantum annealing, on the other hand, is less affected by noise than gate model quantum computing and focuses on the solution of NP-hard problems. This feature enables greater qubit usage and thus more parameters for specific problems. Quantum annealing (QA) is an optimization approach that uses quantum fluctuations to discover the global minimum of a given objective function over a given set of candidate solutions (candidate states). It is mostly employed for issues where the search space is discontinuous (combinatorial optimization problems) with multiple local minima, such as determining the ground state of a spin glass or solving the traveling salesman problem. Apolloni et al. (1990) coined the phrase "quantum annealing" to describe a quantum-inspired classical method in 1988. Quantum annealing can be compared to simulated annealing, in which the "temperature" parameter functions similarly to the tunneling field strength in QA. The temperature determines the likelihood of shifting to a higher "energy"

level from a single current state in simulated annealing. The quantum-mechanical probability of changing the amplitudes of all states in parallel is determined by the strength of the transverse field in quantum annealing. Under some situations, analytical (Morita and Nishimori 2008) and numerical (Santoro and Tosatti 2006) evidence suggests that quantum annealing beats simulated annealing (Heim et al. 2015).

Quantum annealing begins with a quantum-mechanical superposition of all potential states with equal weights. The system then evolves in accordance with the time-dependent Schrödinger equation, which is a natural quantum-mechanical evolution of physical systems. According to the time-dependent strength of the transverse field, which induces quantum tunneling between states, the amplitudes of all candidate states change, resulting in quantum parallelism. If the rate of change of the transverse field is slow enough, the system remains near to the instantaneous Hamiltonian's ground state. If the rate of change of the transverse field is increased, the system may briefly leave the ground state but has a higher possibility of reaching the final problem Hamiltonian's ground state, i.e., diabatic quantum computation. Finally, the transverse field is turned off, and the system is assumed to have arrived to the ground state of the classical Ising model, which corresponds to the solution to the original optimization issue. Immediately following the initial theoretical idea, Kadowaki and Nishimori (1998) an experimental proof of the success of quantum annealing for random magnets was reported. T. Kadowaki and H. Nishimori developed quantum annealing in its current form in "Quantum annealing in the transverse Ising model." D-wave was the first to use a quantum annealing approach to build a quantum computer. D-Wave Systems announced the first commercial quantum annealer on the market, the D-Wave One, in 2011. This system employs a 128-qubit processor chipset.

Compilation, Mapping, and Optimization

Quantum compilation bridges the gap between the computing layer of high-level quantum algorithms and the layer of physical qubits with their specific properties and constraints. Quantum circuit optimization is an essential component of the quantum computing toolchain. Many noisy intermediate-scale quantum (NISQ) devices maintain only loose connectivity between qubits, which means that a valid quantum circuit frequently requires swapping physical qubits to satisfy adjacency requirements. Optimizing circuits to reduce such swaps and other parameters is critical for using quantum hardware in the near future. A significant family of optimal synthesis algorithms functions by completely enumerating all circuits and returning the lowest cost circuit that can do the specified computation; this technique is known as exhaustive or brute-force searching. This method is quite popular in the circuit synthesis community for optimally assembling small frequently used gates or functions to the target gate set, and it can be very effective in these modest instances. Shende et al. (2002) synthesized all minimal gate count circuits for reversible functions on 3 bits using a breadth-first search over the gate set X, CNOT,

T OF. While breadth-first searches are prevalent in reversible circuit synthesis, the lack of efficient unitary representations complicates such approaches. Fowler (2011) avoided this issue by conducting the breadth-first search directly, that is, without the assistance of a pre-computed database with efficient lookup. Non-search-based synthesis has been utilized in quantum computing on occasion. Kliuchnikov et al. (2012) in particular provide an approach for decomposing an arbitrary single-qubit unitary. The earlier described algorithms were largely concerned with lowering gate counts, and any depth reduction was a byproduct of that. However, when there are many computational resources available, it can frequently make sense to raise complexity in order to parallelize operations to take advantage of the extra resources, as in classical computing. However, when there are many computational resources available, it can frequently make sense to raise complexity in order to parallelize operations to take advantage of the extra resources, as in classical computing. Broadbent and Kashefi (2009) develop an algorithm for translating quantum circuits to a pattern (a computation in the measurement-based model) that adds a number of additional ancillas linear in the number of gates. Mapping refers to assigning logical qubits to the physical qubits of the hardware.

In the following section, mapping and optimization specifically for superconducting and trapped-ion quantum computers is discussed.

Superconducting Quantum Computers

Coupling Constraints and Need for SWAP Operation

A qubit in superconducting quantum systems is connected to one or more neighboring qubits using resonators (waveguides) that allow a multi-qubit gate between them. Figure 4b depicts the qubit connectivity graphs for an IBM computer (IBM_lima). The nodes (the circles) represent the qubits. Through coupling graph, it is understood that the native 2-qubit gate (CNOT in IBM and CZ in Rigetti) can only be applied between connected qubits. For instance, CNOT between qubit-1 and 2 is allowed on IBM_lima device as there exists an edge between these qubits in the graph. However, CNOT cannot be applied directly between qubit-1 and 3 as they are not connected. This limited connectivity presents a challenge in quantum circuit mapping and is often referred to as coupling constraints. The constraint is handled by routing qubits via the SWAP operation so that logical qubits with 2-qubit operations become nearest neighbors. Qubit mapping is the term used in the literature to describe the process of changing a quantum circuit to fit hardware restrictions. The final SWAP-inserted version is a nearest-neighbor (NN) compliant circuit that can be run on a quantum computer directly. It's worth noting that the additional SWAP operations must be decomposed to the target hardware's basic gates before being executed. Any traditional qubit mapping procedure entails (a) selecting physical qubits on the hardware for the logical qubits in the circuit (qubit allocation), (b) initial one-to-one mapping of the logical and physical qubits (initial placement), and (c) adding (as few as possible) SWAP operations to meet the hardware constraints for the entire circuit.

Compilation and Optimization

Additional SWAP operations to fulfill the communication need among qubits is an NP-complete problem, according to Siraichi et al. (2018). In the literature, there are two different techniques to reducing the amount of SWAP operations during the qubit mapping procedure. In the first approach, the problem is formulated as an instance of constraint satisfaction problem, and later, powerful reasoning engines (e.g., SMT solver, ILP solver, and SAT solver) are used to find the best possible solution to meet these constraints (Bhattacharjee et al. 2019). Although such approaches frequently yield near-optimal results for small circuits, they are unsuitable for big circuits, resulting in significant compilation time overhead. The second method is based on effective heuristics that gradually lead to a solution (Zulehner et al. 2018; Li et al. 2019). However, decisions at each stage are made with the goal of maximizing the gain in the current step, which may result in sub-optimal results (e.g., local optima). The qubit mapping problem is solved using A* heuristics (Zulehner 2019). The proposed approach chooses a single SWAP operation that minimizes the cumulative SWAP distances of all the two-qubit operations in the current layer in each algorithm iteration. The minimum number of SWAP operations required to produce a logical qubit nearest neighbor to another qubit is known as SWAP distance. The technique is repeated until the cumulative SWAP distance for the current layer reaches zero. The algorithm moves on to the next layer after finding a set of SWAP operations that discovers a new logical-to-physical qubit mapping that meets the hardware restrictions to execute all of the gate operations in the current layer. The look-ahead strategy is a variation of their primary approach that included the cumulative SWAP distances of the following layers in the cost function to determine SWAP operations for the present layer. The look-ahead method frequently yields a superior solution at the cost of increased compilation time. The approach in Zulehner (2019) verifies all the mapped qubits as well as the qubits related to them before adding a SWAP gate. However, it has been reasoned (Li et al. 2019) that this number can be minimized because not all physical qubits in the routing choice have the same "priority." Priority qubits are active qubits in the "front layer" and the qubits related to them. They use a lesser number of SWAPs to examine the SWAP between these reduced sets of qubits for routing. For bigger quantum computer designs and quantum circuits, this reduction can be significant. Murali et al. (2019) demonstrates another routing optimization aspect. The control and target qubits of the CNOT under discussion are reserved in the proposal, and they use one-bends paths for routing.

Trapped-Ion Quantum Computers

Shuttle Operation

A major hurdle in realizing large TI systems is confining many ions in a single trap as it decreases the spacing between ions, making it challenging to pulse a qubit using laser controllers selectively. Moreover, the gate time becomes slow, which results in longer program execution time. Therefore, the pathway to scalability in

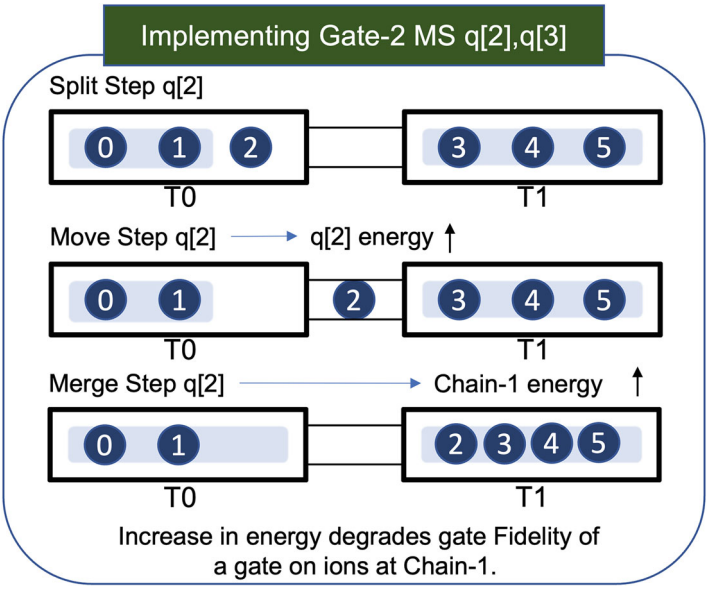


Fig. 9 Shuttle steps to move ion-2 from trap T0 to trap T1

TI systems involves multiple interconnected traps. However, in a multi-trap system, computation is sometimes required on data from ions situated in different traps. For such cases, one ion needs to be shuttled (moved) from one trap to another so that the ions are co-located and the gate operation can be performed. A compiler adds shuttle operations to a quantum program to satisfy the inter-trap communication; however, the shuttle operation increases program execution time and degrades quantum gate fidelity. (The gate fidelity (F) is usually defined as the complement of the error rate. A lower gate fidelity will introduce more errors in the output and can completely decimate the result.) The shuttle operation involves several steps as shown in Fig. 9. For example, to implement gate MS q[0], q[1], it involves ions from the same trap (T0) and can be executed directly. However, to execute gate MS q[2], q[3] involves ions from different traps. Therefore, a shuttle operation is needed to bring both ions into the same trap. For the shuttle operation, first, ion-2 is split from Chain-0 and shuttled from T0 to T1, adding energy to the ion. Then, ion-2 is merged to the Chain-1. Finally, gate MS q[2],q[3] can be executed as the ions are in the same trap (T1). This increase in energy degrades gate fidelity. Therefore, it is essential to minimize the number of shuttle operations.

Compilation and Optimization

Murali et al. (2020b) proposed the first compiler for a multi-trap TI system. They studied trap geometry, trap size, and gate implementation methods in depth. Several compilation policies were also offered, including an initial mapping policy, a shuttle direction policy, and a traffic block resolution policy. Realistic hardware performance parameters are included in the toolchain. It compiles an application (quantum

circuit) to address communication needs by adding more shuttle operations. The architectural policies aim to keep the number of shuttles to a minimum. The tool creates a machine-executable circuit and reports on its dependability, operation count, and execution time. Another recent study on the compilation for TI systems is reported in Wu et al. (2021). They created a compiler for the linear tape model, a TI technology variation. A chain of ions is moved back and forth around a gate zone in the linear tape model to apply gate excitation to different qubits. Because back-and-forth movement of the ion chain increases chain energy, lowers gate quality, and increases execution time, the compiler strives to minimize it. Abdullah Ash Saki et al. in (2022) present compiler optimizations for multi-trap TI quantum computers. Furthermore, the improved compilation enhances the program fidelity up to 22.68X with a modest increase in the compilation shuttles compared to the previous state of the art.

Considerations for Noisy Systems

In quantum systems, single-qubit, multi-qubit gate errors, relaxation and dephasing times, and measurement errors are prominent sources of noise. These errors have varying error rates, implying that some qubits or qubit couplings are less erroneous than others. As a result, noise awareness can be employed in the software stack to optimize a program for a certain hardware to improve program execution dependability. This section reviews noise resiliency research that is both technology neutral and superconducting technology specific.

Technology Agnostic Work

Noise-Aware Qubit Mapping

Some qubits are better (less erroneous) than others at performing computations. Several noise-aware mapping algorithms have been developed as a result of this observation (Ash-Saki et al. 2019; Murali et al. 2019; Tannu and Qureshi 2019). Prioritizing less erroneous qubits to conduct the majority of gate operations is the key strategy. The main approach is prioritizing less erroneous qubits to perform the majority of the gate operations. The authors of Murali et al. (2019) employed the satisfiability modulo theorem (SMT) to make decisions about qubit allocation and movement while accounting for error rate changes. In addition to gate error, they factored readout errors into their allocation choice. Their weighted technique gives users the freedom to choose between gate and measurement errors. Tannu and Qureshi (2019) proposes policies such as variation-aware qubit allocation (VQA) and variation-aware qubit mobility (VQM). To increase the program's success rate, the authors propose harnessing qubit-to-qubit variance. VQA selects a set of physical qubits with the highest cumulative connectivity strength. The cumulative coupling strength represents two things: (a) a qubit is coupled to more neighbors, which is favorable for optimum routing (less SWAP), and (b) the 2-qubit operations between the qubit and its neighbors will be less erroneous. Furthermore, the VQM

policy ensures that the compiler chooses a routing path with fewer erroneous links. In Ash-Saki et al. (2019), the authors present QURE to schedule gate operation to less noisy qubits intelligently, thus resulting in better fidelity of the output state. They propose two approaches: (a) isomorphic subgraph (ISG) search and (b) greedy, to find a better allocation of program (logical) qubits to hardware (physical) qubits. They propose using the ISG search approach to start with an optimal depth version of a quantum circuit and check multiple isomorphic subgraphs systematically. Each subgraph is given an approximate success probability, and the subgraph with the highest success probability is chosen to execute the circuit. They demonstrated that QURE can improve correct output probability or fidelity by a large margin without incurring any physical or circuit-level overhead in a rigorous simulation using a model noisy quantum system and an experiment with IBM's real quantum device.

Measurement Error Mitigation

The final state of a quantum circuit is measured once it has completed its execution. However, due to readout error, reading out a qubit containing a 1 may result in a 0 and vice versa; this arises due to measurement circuitry imperfections. A single-qubit measurement, for example, is performed in two steps in IBM's superconducting quantum hardware: (a) executing the readout pulse on the qubit's readout channel and (b) recording the associated signal, which measures the qubit's energy state, on the acquisition channel. The signal collected during the course of the acquisition is summed to produce a single complex value, which is then plotted in an I-Q plane, with $|0\rangle$ and $|1\rangle$ meant to represent separate clusters. To classify the measured state ($|0\rangle$ or $|1\rangle$) from the imaginary and real components of the complex value, IBM currently utilizes a linear classifier. A synthetic dataset is used to train the classifier. To construct this dataset, the qubit is prepared in the $|0\rangle$ and $|1\rangle$ states many times, followed by measurement operations. The input features to the classifier are the real and imaginary components, and the actual states are the labels. The authors of Patel and Tiwari (2020) demonstrated that the linear classifier has nonuniform measurement errors. When the true state is closer to $|1\rangle$, the error magnifies significantly. The considerable overlap zone generated by the linear decision boundary between the $|1\rangle$ and $|1\rangle$ states contributes to the error magnification. The authors presented two nonlinear classifiers (based on circular and elliptical decision boundaries) and trained them to minimize the variance in measurement errors across different states to get around this problem. The variation of the errors was reduced significantly over the linear classifier measurement, according to the authors.

Superconducting-Specific Work

Crosstalk Mitigation

As two gates run in parallel, the crosstalk error occurs, resulting in an increase in gate faults of two parallel gates when compared to isolated gates. The authors of Murali et al. (2020a) conducted extensive trials on numerous IBM devices

to characterize crosstalk using simultaneous randomized benchmarking (SRB) (Gambetta et al. 2012). They concluded that not all couplings are sensitive to crosstalk, i.e., crosstalk between certain couplings is minimal, whereas crosstalk between some couplings can result in a 2X–3X increase in error rates, and crosstalk disappears after a 1-hop distance. The authors presented a gate scheduling strategy to minimize crosstalk based on crosstalk characterization results, where they serialized the parallel gates at the cost of greater program depth (run-time) and therefore decoherence. To examine the reduction in crosstalk error and rise in decoherence error due to gate serialization, the authors created an SMT-based scheduler (XtalkSched). Crosstalk-aware scheduling has been shown to enhance program integrity by up to 5.6X. Another paper in Ding et al. (2020) attempted dynamic qubit frequency assignment to reduce crosstalk. Qubits are frequency addressable, which means that if two adjacent qubits have sufficiently distinct operating frequencies, crosstalk from one will be reduced. However, the operating frequency range of a qubit is limited, resulting in frequency congestion. The authors suggested a software method that dynamically allocates separate frequencies to surrounding qubits to overcome the frequency crowding problem. When compared to the gate serialization technique, they find a 13.3X increase in program success rate. The solution is applicable to frequency tunable qubits, according to the authors.

Leveraging Extended Native Gates

There are still very few gates that can be realized on quantum hardware. Only a single 2-qubit gate, the CNOT gate, is supported in IBM's superconducting hardware and Mølmer–Sørensen gate in the IonQ's trapped-ion hardware. However, new hardware is emerging which supports multiple 2-qubit gates (Abrams et al. 2020). In general, reducing gate count is desired to minimize decoherence and gate error for better resilience. An extended native gate set can make the gate decomposition step more efficient. In Abrams et al. (2020), the authors note that a SWAP can be decomposed using two gates (1 CZ + 1 iSWAP) when both CZ and iSWAP gates are available as native instructions. However, if only CZ or iSWAP is available as the native gate, it takes 3 CZ/I SWAP to decompose a SWAP. As a result, in NISQ architectures with limited connectivity, an enlarged gate set can dramatically lower the gate count from SWAP insertions. For a test case on QAOA circuits, they report a 30% percent reduction in gate depth.

Application-Specific Compilation

Quantum compilers typically use generic rules to optimize every given quantum program, and they don't take program-specific information into account while doing aggressive optimization. There have been recent papers (Alam et al. 2020a,b,c) that give algorithm-specific compilation approaches for QAOA, which is an outstanding near-term algorithm. The ZZ interactions in QAOA may be accomplished with 2 CNOTs and 1 RZ operation inside a level and are commutative (Alam et al. 2020a), i.e., these operations can be reordered without affecting the circuit's output state. In Alam et al. (2020a), the authors propose several QAOA-specific optimizations, including parallelization of ZZ operations using a binary bin-packing algorithm

(instruction parallelization – IP), repeated compilation of QAOA circuits with reordered layers guided by a branch-and-bound optimization heuristic (iterative compilation), and layer-by-layer circuit construction and compilation prioritizing operations that require fewer SWAPs (incremental compilation). They also discuss various techniques designed to alter QAOA circuit features in order to execute intelligent initial qubit allocation (qubit allocation and initial mapping (QAIM)/variation-aware qubit placement (VQP)). Over the existing state-of-the-art methodologies, these techniques delivered a 53% reduction in circuit depth, a 23% reduction in gate count, and a 63% increase in estimated success probability of QAOA circuits. In addition, the paper also demonstrated about 26% improvement in performance on an actual IBM device.

Conclusion

This chapter introduced the fundamentals of quantum computing architectures. It then went over various quantum hardware technologies that are being researched. The quantum software stack is then reviewed, with an emphasis on compilation, mapping, and optimization heuristics used for two of the most common quantum hardware – superconducting quantum computers and trapped-ion quantum computers. Quantum computing has seen steady growth in recent years. However, in order to make the promise of quantum computing a reality, innovations at various levels of the computing stack are required. Only improving hardware and developing new quantum algorithms will not suffice; a middle stack that bridges the gap between algorithm and hardware is required. As full-fledged error correction will not be possible on NISQ devices, error mitigation is the way to move forward in the foreseeable future. Therefore, this chapter also goes over the various types of errors plaguing the quantum systems and their effect on program performance and reviews recent literature pertaining to optimizing and mitigating the effect of noise on quantum program reliability.

Acknowledgments This material is based upon work supported by NSF (CNS-1814710, DGE-1821766, CNS-2129675, CCF-2210963, DGE-2113839, ITE-2040667), gifts from Intel, and seed grants from Penn State ICDS and Huck Institutes of the Life Sciences.

References

Abhari AJ, Faruque A, Dousti MJ, Svec L, Catu O, Chakrabati A, Chiang C-F, Vanderwilt S, Black J, Chong F (2012) Scaffold: quantum programming language. Technical report, Department of Computer Science, Princeton University

Abrams DM, Didier N, Johnson BR, da Silva MP, Ryan CA (2020) Implementation of XY entangling gates with a single calibrated pulse. Nat Electr 3(12):744–750

Alam M, Ash-Saki A, Ghosh S (2020a) Circuit compilation methodologies for quantum approximate optimization algorithm. In: 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, pp 215–228

Alam M, Ash-Saki A, Ghosh S (2020b) An efficient circuit compilation flow for quantum approximate optimization algorithm. In: 2020 57th ACM/IEEE Design Automation Conference (DAC). IEEE, pp 1–6

Alam M, Ash-Saki A, Li J, Chattopadhyay A, Ghosh S (2020c) Noise resilient compilation policies for quantum approximate optimization algorithm. In: Proceedings of the 39th International Conference on Computer-Aided Design, pp 1–7

Apolloni B, Cesa-Bianchi N, De Falco D (1990) A numerical implementation of "quantum annealing". In: Stochastic Processes, Physics and Geometry: Proceedings of the Ascona-Locarno Conference, pp 97–111

Arute F, Arya K, Babbush R, Bacon D, Bardin JC, Barends R, Biswas R, Boixo S, Brandao FGSL, Buell DA et al (2019) Quantum supremacy using a programmable superconducting processor. Nature 574(7779):505–510

Ash-Saki A, Alam M, Ghosh S (2019) Qure: qubit re-allocation in noisy intermediate-scale quantum computers. In: Proceedings of the 56th Annual Design Automation Conference 2019, pp 1–6

Bhattacharjee D, Saki AA, Alam M, Chattopadhyay A, Ghosh S (2019) MUQUT: multi-constraint quantum circuit mapping on NISQ computers. In: 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). IEEE, pp 1–7

Broadbent A, Kashefi E (2009) Parallelizing quantum circuits. Theor Comput Sci 410(26):2489–2510

Chatterjee A, Stevenson P, De Franceschi S, Morello A, de Leon NP, Kuemmeth F (2021) Semiconductor qubits in practice. Nat Rev Phys 3(3):157–177

Cirac JI, Zoller P (1995) Quantum computations with cold trapped ions. Phys Rev Lett 74(20):4091

Cirq documentation. https://cirq.readthedocs.io/en/stable/

Cross AW, Bishop LS, Smolin JA, Gambetta JM (2017) Open quantum assembly language. arXiv preprint arXiv:1707.03429

Deutsch D, Jozsa R (1992) Rapid solution of problems by quantum computation. Proc R Soc Lond Ser A: Math Phys Sci 439(1907):553–558

Ding Y, Gokhale P, Lin SF, Rines R, Propson T, Chong FT (2020) Systematic crosstalk mitigation for superconducting qubits via frequency-aware compilation. In: 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, pp 201–214

Farhi E, Goldstone J, Gutmann S (2014) A quantum approximate optimization algorithm. arXiv preprint arXiv:1411.4028

Fowler AG (2011) Constructing arbitrary Steane code single logical qubit fault-tolerant gates. Quantum Inf Comput 11(9–10):867–873

Fu P, Kishida K, Ross NJ, Selinger P (2020) A tutorial introduction to quantum circuit programming in dependently typed proto-quipper. In: International Conference on Reversible Computation. Springer, pp 153–168

Gambetta JM, Córcoles AD, Merkel ST, Johnson BR, Smolin JA, Chow JM, Ryan CA, Rigetti C, Poletto S, Ohki TA et al (2012) Characterization of addressability by simultaneous randomized benchmarking. Phys Rev Lett 109(24):240504

Green AS, Lumsdaine PL, Ross NJ, Selinger P, Valiron B (2013) Quipper: a scalable quantum programming language. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp 333–342

Grover LK (1996) A fast quantum mechanical algorithm for database search. In: Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing, STOC'96. Association for Computing Machinery, Philadelphia, pp 212–219

Heim B, Rønnow TF, Isakov SV, Troyer M (2015) Quantum versus classical annealing of Ising spin glasses. Science 348(6231):215–217

Heim B, Soeken M, Marshall S, Granade C, Roetteler M, Geller A, Troyer M, Svore K (2020) Quantum programming languages. Nat Rev Phys 2(12):709–722

Kadowaki T, Nishimori H (1998) Ricottura quantistica nel modello di Ising trasversale. Fis Rev E 58(5):5355

Kjaergaard M, Schwartz ME, Braumüller J, Krantz P, Wang JI-J , Gustavsson S, Oliver WD (2020) Superconducting qubits: current state of play. Annu Rev Condens Matter Phys 11:369–395

Kliuchnikov V, Maslov D, Mosca M (2012) Fast and efficient exact synthesis of single qubit unitaries generated by Clifford and T gates. arXiv preprint arXiv:1206.5236

Lanzagorta M, Uhlmann J (2009) Quantum computer science. Morgan and Claypool Publishers. ISBN:9781598297324

Li G, Ding Y, Xie Y (2019) Tackling the qubit mapping problem for NISQ-Era quantum devices. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pp 1001–1014

Li G, Wu A, Shi Y, Javadi-Abhari A, Ding Y, Xie Y (2021) On the co-design of quantum software and hardware. In: Proceedings of the Eight Annual ACM International Conference on Nanoscale Computing and Communication, pp 1–7

McCaskey AJ, Lyakh DI, Dumitrescu EF, Powers SS, Humble TS (2020) Xacc: a system-level software infrastructure for heterogeneous quantum–classical computing. Quantum Sci Technol 5(2):024002

Montanaro A (2016) Quantum algorithms: an overview. In: NPJ Quantum Information, vol 2, p 1

Morita S, Nishimori H (2008) Mathematical foundation of quantum annealing. J Math Phys 49(12):125210

Murali P, Baker JM, Javadi-Abhari A, Chong FT, Martonosi M (2019) Noise-adaptive compiler mappings for noisy intermediate-scale quantum computers. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pp 1015–1029

Murali P, McKay DC, Martonosi M, Javadi-Abhari A (2020a) Software mitigation of crosstalk on noisy intermediate-scale quantum computers. In: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, pp 1001–1016

Murali P, Debroy DM, Brown KR, Martonosi M (2020b) Architecting noisy intermediate-scale trapped ion quantum computers. In: 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). IEEE, pp 529–542

Nielsen MA, Chuang I (2002) Quantum computation and quantum information. American Association of Physics Teachers

Patel T, Tiwari D (2020) DisQ: a novel quantum output state classification method on IBM quantum computers using openpulse. In: Proceedings of the 39th International Conference on Computer-Aided Design, pp 1–9

Peruzzo A et al (2013) A variational eigenvalue solver on a quantum processor. eprint. arXiv preprint arXiv:1304.3061

Qiskit documentation. https://qiskit.org/documentation/

Qutip documentation. http://qutip.org/documentation.html

Reiher M, Wiebe N, Svore KM, Wecker D, Troyer M (2017) Elucidating reaction mechanisms on quantum computers. Proc Natl Acad Sci 114(29):7555–7560

Rios F, Selinger P (2017) A categorical model for a quantum circuit description language. arXiv preprint arXiv:1706.02630

Ross NJ (2015) Algebraic and logical methods in quantum computation. arXiv preprint arXiv:1510.02198

Saki AA, Topaloglu RO, Ghosh S (2022) Muzzle the shuttle: efficient compilation for multi-trap trapped-ion quantum computers. In: 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, pp 322–327

Santoro GE, Tosatti E (2006) Optimization using quantum mechanics: quantum annealing through adiabatic evolution. J Phys A Math Gen 39(36):R393

Shende VV, Prasad AK, Markov IL, Hayes JP (2002) Reversible logic circuit synthesis. In: Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design, pp 353–360

Shor PW (1994) Algorithms for quantum computation: discrete logarithms and factoring. In: 35th Annual Symposium on Foundations of Computer Science

Shor PW (1999) Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM Rev 41(2):303–332

Siraichi MY, Fernandes dos Santos V, Collange C, Magno Quintão Pereira F (2018) Qubit allocation. In: Proceedings of the 2018 International Symposium on Code Generation and Optimization, pp 113–125

Smith RS, Curtis MJ, Zeng WJ (2016) A practical quantum instruction set architecture. arXiv preprint arXiv:1608.03355

Staq-GitHub. https://github.com/softwareqinc/staq

Steiger DS, Häner T, Troyer M (2018) Projectq: an open source software framework for quantum computing. Quantum 2:49

Strawberry fields. GitHub. https://github.com/xanaduai/strawberryfields

Svore K, Geller A, Troyer M, Azariah J, Granade C, Heim B, Kliuchnikov V, Mykhailova M, Paz A, Roetteler M (2018) Q# enabling scalable quantum computing and development with a high-level DSL. In: Proceedings of the Real World Domain Specific Languages Workshop 2018, pp 1–10

Tannu SS, Qureshi MK (2019) Not all qubits are created equal: a case for variability-aware policies for NISQ-Era quantum computers. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pp 987–999

tket-GitHub. https://github.com/cqcl/pytket

Trauzettel B, Bulaev DV, Loss D, Burkard G (2007) Spin qubits in graphene quantum dots. Nat Phys 3(3):192–196

Treinish M et al (2019) Qiskit: an open-source framework for quantum computing

Wright K, Beck KM, Debnath S, Amini JM, Nam Y, Grzesiak N, Chen J-S, Pisenti NC, Chmielewski M, Collins C et al (2019) Benchmarking an 11-qubit quantum computer. Nat Commun 10(1):1–6

Wu X-C, Debroy DM, Ding Y, Baker JM, Alexeev Y, Brown KR, Chong FT (2021) Tilt: achieving higher fidelity on a trapped-ion linear-tape quantum computing architecture. In: 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA). IEEE, pp 153–166

Zulehner A (2019) Evaluating the flexibility of a* for mapping quantum circuits. In: Thomsen MK, Soeken M (eds) Reversible computation. Springer International Publishing, Cham, pp 171–190

Zulehner A, Paler A, Wille R (2018) An efficient methodology for mapping quantum circuits to the IBM QX architectures. IEEE Trans Comput-Aided Design Integr Circuits Syst 38(7):1226–1236

Design and Tool Solutions for Monolithic Three-Dimensional Integrated Circuits

22

Kyungwook Chang and Sung Kyu Lim

Contents

Introduction . . . 752
Monolithic 3D IC Design Flow . . . 753
Motivation and Background . . . 753
Benefit Trends of Monolithic 3D ICs Across Technology Nodes . . . 754
A Design-Aware Partitioning Approach to Monolithic 3D IC with 2D Commercial Tools . . . 761
Power Supply Integrity of Monolithic Three-Dimensional Integrated Circuits . . . 773
Motivation and Background . . . 773
System-Level Power Delivery Network Analysis for Monolithic 3D ICs . . . 774
Monolithic 3D ICs for Deep Neural Network Hardware . . . 785
Motivation and Background . . . 785
Impact of Monolithic 3D ICs on On-Chip Deep Neural Networks Targeting Speech Recognition . . . 786
Conclusion . . . 801
References . . . 802

Abstract

This chapter is to identify the benefits and challenges of monolithic three-dimensional integrated circuits and to introduce physical design and tool solutions to address the challenges. The physical design and tool challenges of monolithic three-dimensional integrated circuits are addressed with the following

K. Chang (✉)
Suwon, South Korea
e-mail: k.chang@skku.edu

S. K. Lim
Atlanta, USA
e-mail: limsk@ece.gatech.edu

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_65

three categories: design flow, power supply integrity, and application of monolithic three-dimensional integrated circuits.

In design flow, an approach to implement monolithic three-dimensional integrated circuits is introduced. Power supply integrity issues of monolithic three-dimensional stacking technology are addressed. Lastly, deep neural network hardware using monolithic three-dimensional integrated circuits is presented as implementing low-power and high-performance deep neural network hardware is known to be difficult albeit they are widespread and powerful in recognition tasks.

Keywords

Monolithic three-dimensional integrated circuits (Monolithic 3D ICs) · Design automation · Physical design · Reliability

Introduction

As technology scaling faces its physical limits in channel length scaling, degrading process variations, lithography constraints, increased parasitics, and rising manufacturing costs, monolithic three-dimensional (M3D) stacking technology takes center stage in continuing Moore's law. In M3D stacking technology, the devices are fabricated onto multiple tiers sequentially with nanosized monolithic inter-tier vias (MIVs), which connect the topmost metal layer of the bottom tier and the bottommost metal layer of the top tier as shown in Fig. 1. Because MIVs are extremely small, they can achieve much higher vertical integration density and lower resistive-capacitive (RC) parasitics compared to through-silicon vias (TSVs). Owing to the enhancement of fabrication technology, one can harness the true benefit of M3D integrated circuits (ICs) with fine-grained vertical integration (Okada et al. 2014).

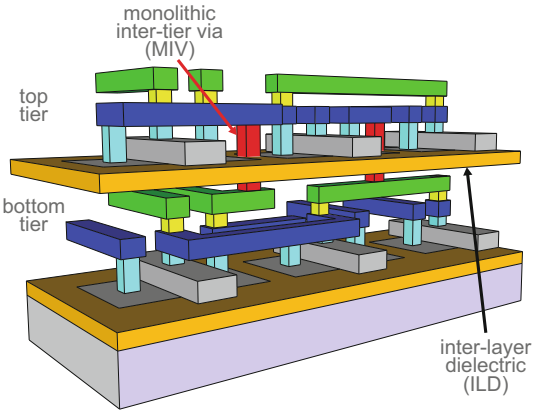


Fig. 1 An example of monolithic three-dimensional (M3D) integrated circuit (IC) structure

M3D ICs show manifold advantages over conventional two-dimensional (2D) ICs by utilizing short vertical connections instead of using long wires in the xy-plane, offering lower power consumption and higher performance. However, the 3D nature casts physical design and tool challenges.

First, as current commercial electronic design automation (EDA) tools do not support 3D placement of cells, most of 3D ICs including TSV-based 3D ICs are implemented by designing each tier separately with tight-timing margins on the signals crossing tiers. However, using separate designs for multiple tiers is prone to non-optimized design. Therefore, several recent studies present methodologies to implement M3D ICs using commercial 2D tools including multiple tiers in a single design.

Then, power supply integrity issues of M3D stacking technology are addressed. Multiple device layers and reduced footprint make M3D ICs suffer from higher power density, which in turn raise voltage drop on power rails of M3D ICs. In order to prevent functional failures and performance degradation due to worsened power supply integrity with lower supply voltage, faster operating clock frequency, and higher power density, power delivery networks (PDNs) of M3D ICs should be analyzed.

Lastly, the challenges in deep neural network (DNN) hardware and the impact of M3D ICs are examined as an application because implementing low-power and high-performance DNN hardware is known to be difficult albeit they are widespread and powerful in recognition tasks. In order to resolve the high demand on computing and memory usage, and an extensive wire connections among neurons, studies, which encompass the influence of key parameters in DNN hardware including DNN architecture choices and underlying workloads as well as the physical design optimization, are presented.

Monolithic 3D IC Design Flow

Motivation and Background

The industry has transitioned from planar MOSFETs to 3D FinFETs at the 14/16 nm node to combat worsening electrostatics and degraded short channel effects due to channel length scaling. Improved transistor characteristics in FinFETs are achieved at the cost of higher parasitic capacitance associated with the 3D fins and the introduction of the local interconnects that are needed to contact the devices to metal routing layers. Due to limited viable transistor options beyond FinFETs and the increasing cost and complexity of lithography strategies to print sub-7 nm node features, traditional Moore's law scaling is slowing down. These limitations create a technology inflection point for "More than Moore" technologies (Arden et al. n.d.) such as M3D ICs to bring value and be adopted into mainstream designs.

To be deployed in real-word designs, M3D ICs need to be cost-effective and deliver power or performance improvement of the order of magnitude similar to that obtained by Moore's law process scaling. Evaluating cost-effectiveness is

non-trivial as M3D stacking technology is still under active research and development. Hence, the power improvement of M3D ICs in an in-order, 32-bit application processor is evaluated while assessing whether or not that improvement is independent of the underlying technology node.

Currently EDA tools do not support M3D ICs, and hence, previous studies have explored implementation approaches of M3D ICs using 2D commercial tools. In Panth et al. (n.d.), in order to estimate cell placement and wire-length of an M3D IC, the dimensions of cells and wires are shrunk, and a shrunk-2D design is implemented in half footprint of the 2D IC. However, using shrunk-2D designs are prone to inaccurate buffer insertion because of inaccurate wire-load estimation (Chang et al. 2017). Moreover, the flow is completely design-agnostic, utilizes very large number of MIVs, and hence, partitions local cells into separate tiers resulting in a non-optimal tier partition. Another M3D IC design flow is proposed in Billoint et al. (2015), which folds 2D placement at the center of the die into two separate tiers. However, using their design flow shows marginal wire-length savings and no power savings and does not take into account design details to guide partitioning, resulting in a non-optimal solution. Therefore, a new M3D IC design flow is necessitated which incorporates design and micro-architecture information during partitioning cells on multiple tiers while supporting accurate buffer insertion with accurate wire-load estimation.

Benefit Trends of Monolithic 3D ICs Across Technology Nodes

First, a comprehensive study investigating the power impact of M3D ICs across technology nodes is presented using a commercial in-order 32-bit application processor on foundry 28 nm, foundry 14/16 nm and 7 nm technology nodes. Based on the observation, M3D stacking technology provides maximum power savings at the 28 nm technology node, and the benefits improve at higher clock frequencies with the reduction of standard cell area in addition to wire-length savings. An in-depth analysis of the results and guidelines for M3D ICs are presented to support the observations.

Analysis on Benefits of Monolithic 3D ICs

Technology Nodes and Design Libraries

Table 1 shows the representative metrics of each technology node used in this experiment, based on previous publications (Yang et al. 2011; Wu et al. 2013; Song et al. 2015; Seo et al. 2014). The 28 nm technology node is planar transistor based, while 14/16 nm is the first-generation foundry FinFET technology node. For these technology nodes, production-level standard cell libraries are utilized containing over 1000 cells and memory macros that were designed, verified, and characterized using foundry process design kit (PDK).

The 7 nm PDK contains electrical models (BSIM-CMG), design rule checking (DRC), layout versus schematic (LVS), extraction and library exchange format

Table 1 Key metrics of foundry 28 nm, 14/16 nm, and 7 nm technology nodes used in the designs compared to 5 nm technology node

Parameters	28 nm	14/16 nm	7 nm	5 nm
Transistor type	Planar	FinFET	FinFET	FinFET
V_{DD}(V)	0.9	0.8	0.7	0.7
CPP (nm)	110 ∼ 120	78 ∼ 90	50	48
M1 pitch (nm)	90	64	36	30
MIV cross-section (nm)	80 × 80	40 × 40	32 × 32	28 × 28
MIV height (nm)	140	170	170	170

(LEF) files.[1] The transistor models incorporate scaled channel lengths and fin-pitches and increased fin-heights compared to previous technology nodes in order to improve performance at lower supply voltages. Multiple threshold voltages (V_T) and variation corners are supported in the 7 nm PDK. Process metrics such as gate pitch and metal pitches are linearly scaled from previous technology nodes, and the design rules are created considering lithography challenges associated with printing these pitches. The interconnect stack is modeled based on similar scaling assumptions. The 7 nm standard cell libraries and memory macros are designed from the scratch and characterized using the PDK.

The M3D IC requires six metal layers on both top and bottom tiers. The MIVs connect M6 of the bottom tier with M1 of the top tier. The size of the MIVs is limited to be 2× the minimum via size allowed in the technology node to reduce MIV resistance. The MIV heights take into account the fact that the MIVs need to traverse through inter-tier dielectrics and transistor substrates to contact to M1 on the top tier. The MIV height increases from 28 nm to 14/16 nm and 7 nm technology nodes because of the introduction of local interconnect middle-of-line (MOL) layer in the sub-20 nm nodes.

Since M3D IC fabrication is done sequentially, high-temperature front-end device processing of the top tier can adversely affect the interconnects in the bottom tier, while low-temperature processing will result in inferior top-tier transistors. Recent work reporting low-temperature processes that achieve similar device behavior across both tiers have been presented (Batude et al. 2015), and hence, all implementations are done with the assumption of similar device characteristics in both tiers.

Implementation Methodology

The standard cell libraries and memory macros for the 28 nm, 14/16 nm, and 7 nm technology nodes are used to synthesize, place, and route the full-chip design. 2D and M3D ICs of the application processor are implemented sweeping the target frequency from 500 MHz to 1.2 GHz in 100 MHz increments across the three

[1] During the writeup of the manuscript, 7 nm academic PDK was not available, thus the need for our own development.

technology nodes. M3D ICs are implemented using shrunk-2D design flow (Panth et al. n.d.). Full-chip timing is met at the appropriate corners (i.e., slow corner for setup and fast corner for hold). Power is reported at the typical corner. The floorplan of the design is customized for each technology node to meet timing, but kept constant during frequency sweeps. Multiple iterations of the 2D and M3D IC floorplan are required at each node to ensure that the designs meet timing. The chip area is fixed such that the final cell utilization is similar across technology nodes.

Power Saving Trend of Monolithic 3D ICs

Figure 2 shows the GDS layouts of the 2D and M3D IC implementation of a 32-bit application processor running at 1.1 GHz.[2] The implementation tools are unable to meet timing at 1.2 GHz target frequency for the 28 nm and 14/16 nm designs, hence their results are reported up to 1.1 GHz, while the 7 nm results up to 1.2 GHz.

The normalized total power consumption of the 2D and M3D ICs across technologies are shown in Fig. 3. The total power of both the 2D and M3D ICs increases with increasing frequency across technology nodes, which is expected. The power saving with the M3D ICs over the 2D ICs is shown in Fig. 4. There are two important trends in Fig. 4: (1) 28 nm node shows the maximum power savings in M3D IC across all frequencies, and (2) the power saving of M3D stacking technology over 2D ICs (i.e., M3D power savings) increases with increasing target frequency of the designs.

To interpret and analyze the results, the following equation is used, which describes the components of dynamic power in an IC.

$$\begin{aligned} P_{\text{dyn}} &= P_{\text{INT}} + \alpha \cdot \left(C_{\text{pin}} + C_{\text{wire}}\right) \cdot V_{\text{DD}}^2 \cdot f_{\text{clk}} \\ &= P_{\text{INT}} + \alpha \cdot \left(r_{\text{p2w}} \cdot C_{\text{wire}} + C_{\text{wire}}\right) \cdot V_{\text{DD}}^2 \cdot f_{\text{clk}}, \end{aligned} \tag{1}$$

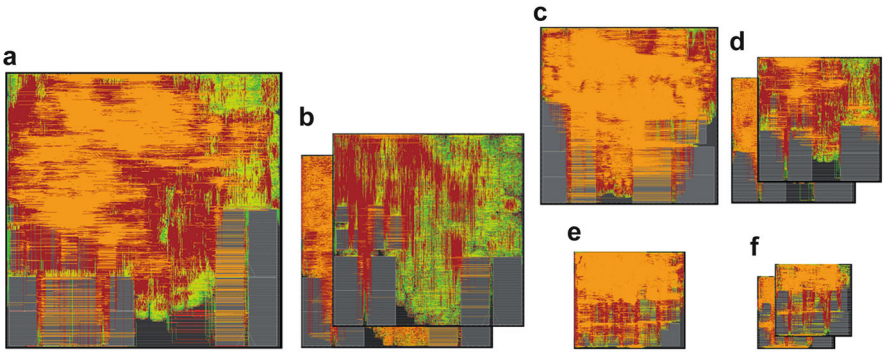


Fig. 2 GDS layouts of (**a**) 28 nm 2D, (**b**) 28 nm M3D, (**c**) 14/16 nm 2D, (**d**) 14/16 nm M3D, (**e**) 7 nm 2D, and (**f**) 7 nm M3D ICs of the application processor at 1.1 GHz

[2]We acknowledge the contribution of ARM for their donation of a commercial 32-bir processor architecture for this research.

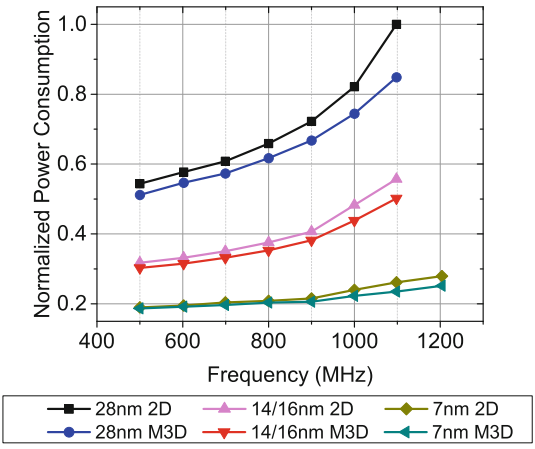


Fig. 3 Normalized total power consumption of 2D and M3D ICs in 28 nm, 14/16 nm, and 7 nm technology nodes

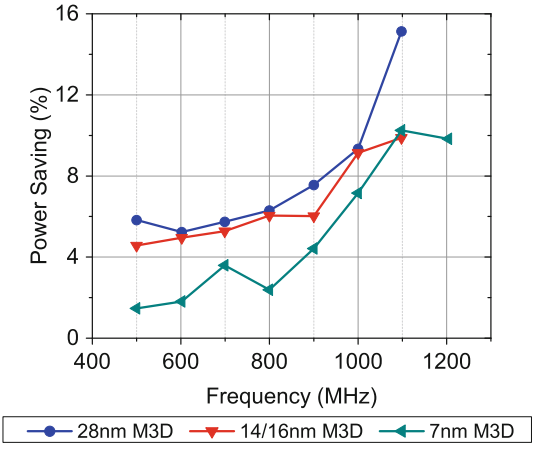


Fig. 4 Power saving of M3D ICs over 2D ICs in 28 nm, 14/16 nm, and 7 nm technology nodes

where P_{INT} is the cell internal power, and the second term describes net switching power where C_{pin} and C_{wire} are the pin and wire capacitance in the design. r_{p2w} is the ratio of the pin capacitance to the wire capacitance. The primary advantage of M3D ICs comes from wire-length reduction resulting in reduced wire capacitance switching power dissipation. With the reduction in wires, the synthesis, place, and route tools can also reduce the drive-strengths of the gates and buffers used to meet the design targets leading to reduced internal power (P_{INT}) and pin capacitance switching component as well. The total power reduction in an M3D IC depends on wire-length reduction, the number of cells and cell size reduction, the ratio of pin capacitance to wire capacitance, and net switching power to internal power in the 2D IC.

Further extending Eq. 1, as internal power and pin capacitance depend on standard cell area, and wire-length affects wire capacitance, M3D power savings can be denoted as follows:

$$\Delta P_{\text{dyn}} = \Delta_{\text{cell}} \cdot \left(P_{\text{INT}} + \alpha \cdot r_{\text{p2w}} \cdot C_{\text{wire}} \cdot V_{\text{DD}}^2 \cdot f_{\text{clk}} \right) + \Delta_{\text{wire}} \cdot \alpha \cdot C_{\text{wire}} \cdot V_{\text{DD}}^2 \cdot f_{\text{clk}}, \tag{2}$$

where Δ_{cell} denotes the standard cell area saving from M3D ICs over the 2D counterparts, and Δ_{wire} denotes the wire-length saving in the M3D IC. This simple linear model gives useful insight in explaining the power saving trends across technology nodes and frequencies.

Analysis of Trends

As can be seen from Fig. 5, at a given frequency, the wire-length saving (Δ_{wire}) as well as the standard cell area saving (Δ_{cell}) is nearly the same across all the three technology nodes.

As the clock frequency is swept, the wire-length saving (Δ_{wire}) does not vary by a large magnitude, ranging between 20% and 25% as shown in Fig. 5. However, with increasing clock frequency, 2D ICs utilize more buffers and higher drive-strength cells to meet timing, whereas M3D ICs can meet timing with lesser number of buffers and lower drive-strength cells because of the wire-length saving. Hence, the standard cell area saving (Δ_{cell}) increases from 2% up to 10 $\sim$ 12% with increasing frequency. With these observations, Eq. 2 is modified to denote Δ_{cell} as a function of f_{clk} in order to reflect the impact of frequency on standard cell area savings.

$$\Delta P_{\text{dyn}} = \Delta_{\text{cell}}(f_{\text{clk}}) \cdot \left(P_{\text{INT}} + \alpha \cdot r_{\text{p2w}} \cdot C_{\text{wire}} \cdot V_{\text{DD}}^2 \cdot f_{\text{clk}} \right) + \Delta_{\text{wire}} \cdot \alpha \cdot C_{\text{wire}} \cdot V_{\text{DD}}^2 \cdot f_{\text{clk}} \tag{3}$$

M3D Power Saving at Low Frequency

At low frequencies (500 MHz), Δ_{cell} is small (= 2.5%) while Δ_{wire} is much higher. Hence, most of the power saving in M3D ICs comes from reduction in wire

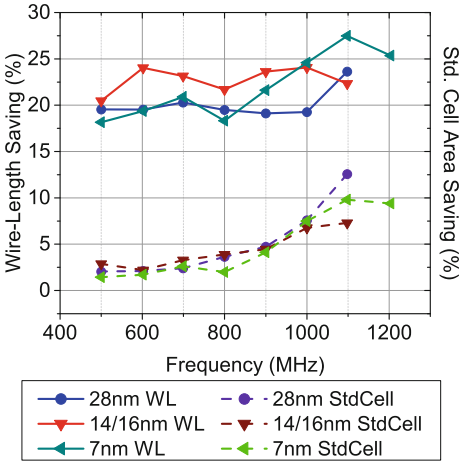


Fig. 5 Impact of M3D ICs on the wire-length (*solid lines*) and standard cell area (*dotted lines*) savings over 2D ICs in 28 nm, 14/16 nm, and 7 nm technology nodes

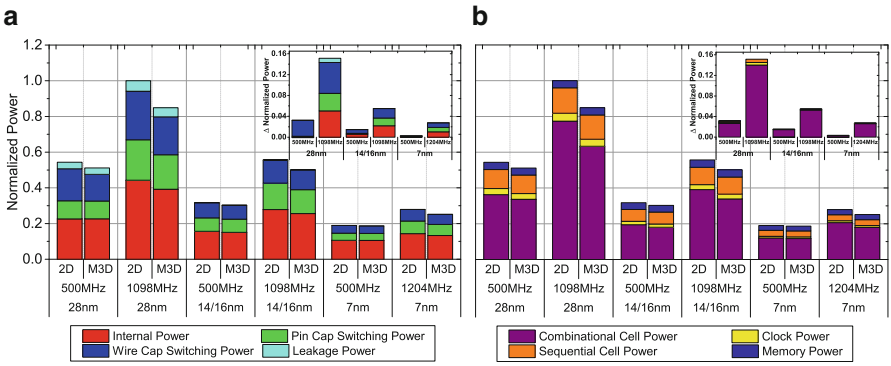


Fig. 6 Power breakdown into (**a**) internal power, pin capacitance switching power, wire capacitance switching power, and leakage power, (**b**) combinational cell power, clock power, sequential cell power, and memory power at the minimum and maximum frequencies of each technology node. The inset plots show the power reduction of M3D ICs for each power component

capacitance switching power. Figure 6a shows the normalized power components of the 2D and M3D ICs across technology nodes at the minimum (500 MHz) and maximum (1.1/1.2 GHz) frequencies. This figure clearly shows that internal power is the dominant portion of the total power accounting for nearly 50% of the total across all frequencies and technology nodes. The rest is split between pin capacitance switching and wire capacitance switching power with leakage power taking up the smallest portion. It is important to note that the pin capacitance switching power and internal power are both related to the number and size of gates used in the design. Power saving due to reduction in wire capacitance switching power is determined by r_{p2w} in the design. Hence, even with $20 \sim 25\%$ wire-length reduction, the total power saving at low frequencies ranges between 1.5% (7 nm technology node) and 6% for (28 nm technology node) because wire capacitance switching power is a small portion of the total power. The 28 nm M3D IC has better power saving at 500 MHz because it has a larger wire capacitance to pin capacitance ratio as shown in Fig. 7.

This difference in pin capacitance versus wire capacitance from 28 nm to 14/16 nm node can be attributed to the difference in gate capacitance associated with planar MOSFETs and 3D FinFETs. FinFET-based technologies have higher gate capacitance due to the 3D fin structure and the introduction of local interconnect MOL layers that contact the device terminals to M1. Therefore, planar MOSFET-based designs are more likely to benefit from M3D ICs compared to FinFET-based designs at advanced nodes.

Another point to note is that with process scaling, wire RC parasitic, especially resistance, increases per unit length. Improving drive-strength of transistors at advanced nodes like 7 nm is extremely challenging. As the ratio of transistor drive versus wire-load decreases at scaled nodes, implementation tools end up using

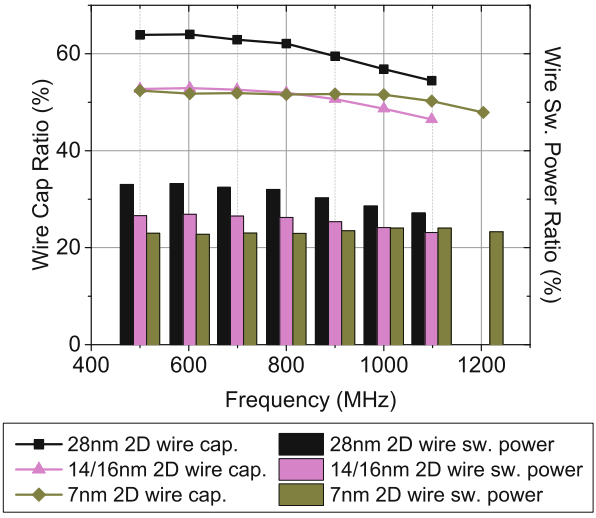


Fig. 7 Wire capacitance to total capacitance ratio, and net switching power to total power ratio in 2D ICs in 28 nm, 14/16 nm, and 7 nm technology nodes

larger cells to drive the same wire-length, hence, effectively increasing r_{p2w}. Hence, technologies with larger transistor fanouts will benefit more from M3D ICs.

M3D Power Saving at High Frequency

At high operating frequencies, as Δ_{cell} increases, it affects both pin capacitance switching power and internal power. As evident from Fig. 6, internal power and pin capacitance switching power can contribute up to 70% of the total power of the 2D IC at high frequencies. Hence, the total power savings at maximum frequencies approach 10% or more as M3D ICs benefit from reduction in all power components, predominantly internal power and pin capacitance switching power.

In order to understand the impact of frequency on M3D power savings, a hypothetical scenario is considered when, with increasing clock frequency, $\Delta_{cell}(f_{clk}) = \Delta_{wire}$. At this frequency point, Eq. 3 can be modified to the following expression.

$$\Delta P_{dyn} = \Delta_{wire} \cdot \left(P_{INT} + \alpha \cdot C_{tot} \cdot V_{DD}^2 \cdot f_{clk} \right), \tag{4}$$

where C_{tot} is total capacitance ($C_{pin} + C_{wire}$) of a 2D IC. At this clock frequency, the M3D power saving does not depend on r_{p2w}. Moreover, as discussed previously, P_{INT} being the dominant component of the total power, M3D power saving depends more on Δ_{cell} and the ratio of internal power versus net switching power than Δ_{wire} or r_{p2w}.

Figure 6b shows power breakdown according to the type of cells. As the number of hard macros (e.g., memory blocks) and sequential cells is fixed, the power

Table 2 Normalized iso-performance design and power metric comparison of 2D and M3D ICs with application processor in 28 nm, 14/16 nm, and 7 nm technology nodes. All values are normalized to corresponding 28 nm 2D parameters. Capacitance and power values are normalized to 28 nm 2D total capacitance and 28 nm 2D total power, respectively

Parameters	28 nm	14/16 nm	7 nm	28 nm	14/16 nm	7 nm
Footprint	1 × 1	0.64 × 0.64	0.41 × 0.35	−51.1%	−50%	−54.7%
Density	1	0.899	0.803	−10.9%	−8.9%	−12.3%
Cell count	1	1.029	1.251	−7.8%	−7.3%	−9.5%
Std. cell area	1	0.32	0.085	−12.6%	−7.3%	−9.8%
Wire-length	1	0.649	0.437	−23.6%	−22.3%	−27.5%
Wire cap	0.544	0.328	0.207	−23.3%	−13.1%	−13.2%
Pin cap	0.456	0.378	0.205	−16.5%	−9.1%	−12%
Total cap	1	0.706	0.412	−20.2%	−11%	−12.6%
Internal power	0.443	0.278	0.136	−11.4%	−7.9%	−8.6%
Wire cap switching power	0.271	0.129	0.063	−21.8%	−14%	−12.7%
Pin cap switching power	0.227	0.148	0.062	−14.9%	−10.1%	−11.5%
Leakage power	0.059	0.001	0.001	−13.4%	−5%	−3.2%
Total power	1	0.557	0.262	−15.1%	−9.9%	−10.3%

consumed by these cells does not change in M3D ICs. On the other hand, power consumed by combinational cells and clock signal can be reduced effectively in M3D ICs utilizing lower number of buffers and using lower drive-strength cells.

Table 2 shows all the important design metrics of both 2D and M3D ICs across foundry 28 nm, 14/16 nm, and 7 nm technology nodes at 1.1 GHz. Since 1.1 GHz is the maximum frequency for 28 nm and 14/16 nm implementation, and the second highest for 7 nm design, the significant standard cell area saving as well as wire-length saving is achieved with M3D ICs.

Since the operating clock frequency is high, M3D ICs save the standard cell area by 9.9% on average for the three implementations, resulting in the internal power and pin capacitance switching power savings. Although the ratios of internal power and pin capacitance switching power on 2D ICs versus M3D ICs (11.4% and 14.9% in the 28 nm designs) is smaller than wire capacitance switching power ratio (= 21.8%), since those components account for more than 70% of the total power, they have a bigger contribution to the total power savings.

A Design-Aware Partitioning Approach to Monolithic 3D IC with 2D Commercial Tools

Based on the observations in the previous section, a new methodology called "cascade2D design flow" to implement M3D ICs using 2D commercial tools is presented in this section. Cascade-2D design flow utilizes a design-aware partitioning scheme where functional modules with very large number of connections are partitioned into separate tiers.

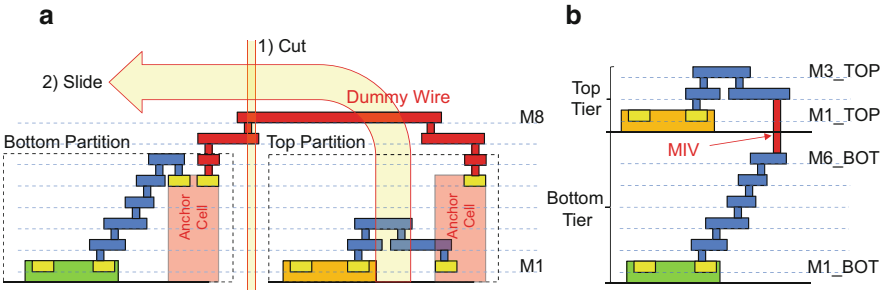


Fig. 8 M3D IC design scheme of cascade-2D design flow. (**a**) A cascade-2D design implementation with a set of anchor cells and dummy wires which models MIVs, and (**b**) the equivalent M3D IC

In this flow, MIVs are modeled as sets of anchor cells and dummy wires, which enable to implement and optimize both top and bottom tiers simultaneously in a 2D IC. Cascade-2D design flow reduces standard cell area effectively, resulting in significantly better power savings than shrunk-2D design flow. Experimental results show that M3D ICs implemented with cascade-2D design flow (i.e., cascade-2D M3D ICs) can achieve up to 4× better power savings compared to those with shrunk-2D design flow (i.e., shrunk-2D M3D ICs), while using an order of magnitude less MIVs. In the best-case scenario, cascade-2D M3D ICs result in 25% higher performance at iso-power and up to 20% power reduction at iso-performance compared to 2D ICs. Additionally, by leveraging smaller standard cells, M3D ICs can save up to 10% die area which directly translates to reduced costs.

Figure 8 shows the "cut-and-slide" methodology of cascade-2D design flow with sets of anchor cells and dummy wires. As can be clearly seen, the anchor cells and dummy wires model MIVs and the cascade-2D design implementation in Fig. 8a are functionally equivalent to the M3D IC in Fig. 8b.

Implementation Methodology

Table 3 presents a qualitative comparison of cascade-2D design flow with shrunk-2D design flow. Figure 9 shows the flow diagram of this methodology. First, functional blocks are partitioned into two groups, the top and bottom group, creating signals crossing the two groups, which become MIVs in an M3D IC. Then, the location of MIVs are determined, and lastly, a cascade-2D design is implemented with sets of anchor cells and dummy wires in 2D space, which is equivalent to the final M3D IC.

Design-Aware Partitioning Stage

In this step, an RTL is partitioned into two groups, the top and bottom group, which represent the top and bottom tier of an M3D IC, respectively. The partition can be performed in two ways: (1) based on the organization of the design micro-architecture and (2) by extracting design information from the 2D implementation.

Table 3 Qualitative comparison of cascade-2D design flow and shrunk-2D design flow

Cascade-2D design flow	Shrunk-2D design flow
Support block- and gate-level M3D ICs	Support only gate-level M3D ICs
Capable of handling RTL-level constraints	Cannot handle RTL-level constraints
Highly flexible; can implement any partitioning algorithm	Implements area-balanced min-cut algorithm for partitioning cells
Designer has complete control over tier-assignment of cells/blocks	Designer controls bin-size but not actual tier-assignment of gates
Implements top and bottom tier in a single design	Implements top and bottom tier separately
Buffer insertion based on actual technology parameters	Buffer insertion based on shrunk technology parameters

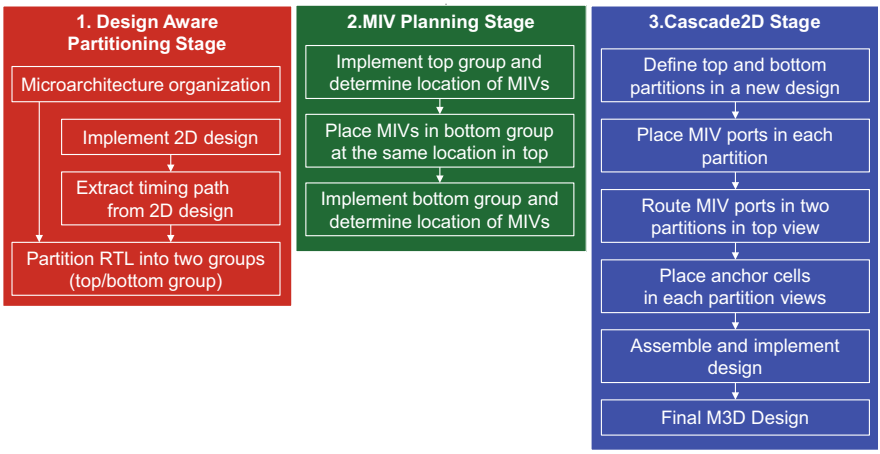


Fig. 9 Flow diagram of cascade-2D design flow

Because M3D ICs offer vertical integration of cells, power and performance improvement is achieved by placing inter-communicating functional modules separated by a large distance in the xy-plane in a 2D IC on separate tiers and reducing the distance by utilizing the z-axis in an M3D IC. With a detailed understanding of the micro-architecture organization, functional modules can be pre-partitioned into separate tiers. For example, consider two functional modules whose connecting signals have a tight-timing budget (e.g., a data path unit and its register bank). Placing these modules into separate tiers and connecting them with MIVs can help reduce the wire-length.

In case it is non-trivial to partition based on the understanding of micro-architectural organization, the design information from a 2D implementation can be utilized to help guide the partitioning process. By extracting timing paths from a 2D IC, the number of timing paths crossing each pair of functional modules can be quantified, which is called "degree of connectivity" between functional modules.

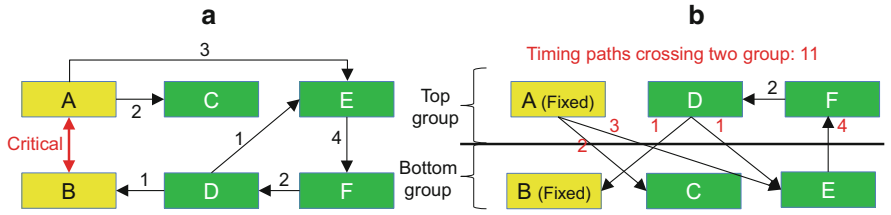


Fig. 10 An example of the design-aware partitioning scheme of cascade-2D design flow. (**a**) Pre-partitioned modules (*yellow box*), and degree of connectivity (numbers on the arrows) of rest of modules (*green box*). (**b**) Result of the design-aware partitioning

The standard cell area of each functional module is also extracted from the 2D IC to balance cell area between the tiers.

After obtaining the degree of connectivity of functional modules and their cell area, the design is partitioned into two groups based on the following criteria:

- Balance the cell area of the top and bottom group.
- Maximize the number of timing paths crossing the two groups.

These criteria help (1) the functional blocks, which have a very high degree of connectivity, be placed onto separate tiers and minimize the distance between them, and (2) balance the standard cell area of the two tiers. Figure 10 shows an example of design-aware partitioning. Modules A and B are fixed on two different groups based on organization of the design micro-architecture, modules C, D, E, and F are partitioned maximizing the number of timing paths crossing two groups and balancing cell area of two groups. However, cascade-2D design flow is extremely flexible and can incorporate any number of constraints for partitioning cells or modules into separate tiers. Depending on the type of design, the designer may wish to employ different partitioning criteria than presented here and the subsequent steps (MIV Planning Stage and Cascade-2D Stage) would remain the same. Hence, this flow is an ideal platform to evaluate different tier partitioning schemes for M3D ICs.

At this stage, it is important to understand that there are two types of IO ports in the design. There are a set of IO ports that were created because of the "design-aware partitioning" step. These IO ports connect the top and bottom groups of the design, and they are referred as MIV ports since they eventually become MIVs in an M3D IC. Additionally, there exist a set of IO ports for the top-level pre-partitioned design. These are same as the conventional IO ports of 2D ICs.

MIV Planning Stage

After partitioning an RTL into the top and bottom groups, the location of MIVs is determined. First, the top group is implemented, and MIVs ports are placed above their driving or receiving cells on the top routing metal layer, so that wire-length between MIV ports and relevant cells is minimized. As explained previously, MIV ports are actually IO ports that connect the top and bottom groups. Leverage the

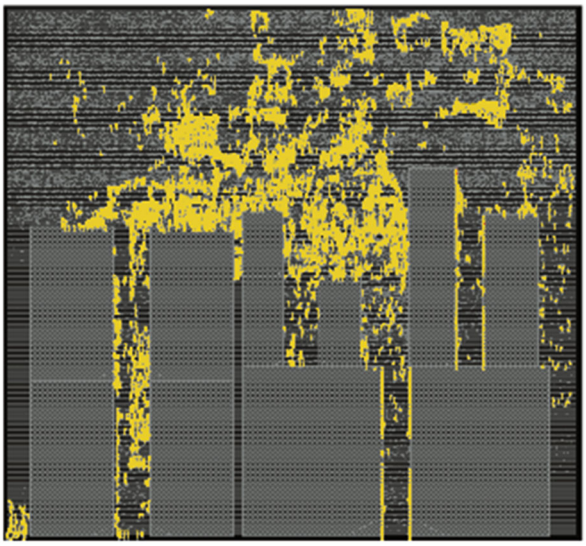


Fig. 11 Location of MIVs (*yellow dots*) after completing MIV planning stage in cascade-2D design flow

fact that all cell placement algorithms in commercial EDA tools tend to place cells close to IO ports to minimize timing, the bottom group is implemented using the location of MIVs determined from the top group implementation. In this way, the cell placement of the top group guides the cell placement of the bottom group using the pre-fixed MIV ports.

The IO ports of the top-level design are assumed to be connected only to the top tier in M3D ICs. Therefore, it is possible that some IO signals need to be directly connected to functional modules in the bottom group. These feed-through signals will not have any driving or receiving cells on the top group. Hence, the MIV ports for those signals cannot be placed with top group implementation and are determined during the bottom group implementation.

Figure 11 shows the location of MIVs after implementing the bottom group. After obtaining the location of complete set of MIVs, standard cell placement in the top and bottom group implementation is discarded, and only the MIV locations are retained.

Cascade-2D Stage

In this step, a cascade-2D design is implemented, which models an M3D IC in a single 2D design with sets of anchor cells and dummy wires, using partitioning technique supported in Cadence® Innovus™.

First, a new die with both tiers placed side by side is created, with the same total area as the original 2D design. Top and bottom partitions are defined in the die, and a hard fence for placement is set, so that cells in the top partition are placed only on the top half of the die, and cells in the bottom partition only on the bottom half of the die. Then, two hierarchies of the design are created as follows:

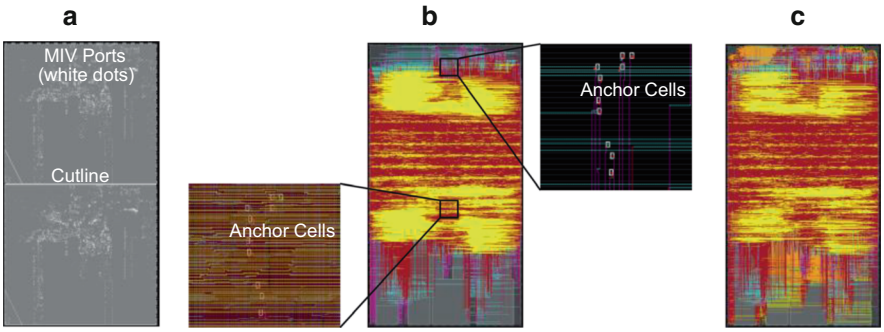


Fig. 12 GDS layouts in each step in cascade-2D design flow. (**a**) Top view after placing pins for MIVs, (**b**) after assembling top view and top- and bottom-partition view, (**c**) after implementing cascade-2D designs

- First level of hierarchy: Top view, which contains only two cells, top-partition cell and bottom-partition cell. These two cells contain pins which represent MIVs for the top and bottom tier, respectively.
- Second level of hierarchy: Top-partition cell, which contains the top-partition view where standard cells from the top group are placed and routed.
- Second level of hierarchy: Bottom-partition cell, which contains the bottom-partition view where standard cells from the bottom group are placed and routed.

In the top view, pins are placed, representing MIVs, in the top-partition cell and bottom-partition cell on the top routing metal layer (i.e., M6 in Fig. 8). The pin locations are the same as the MIV location derived in MIV Planning Stage. Figure 12a shows placed pins for MIVs in the top view.

Then, using 3 $\sim$ 4 additional metal layers above the top routing metal layer used in the actual design, (i.e., M7 $\sim$ M8 in Fig. 8), the pins on the top-partition cell and bottom-partition cell are routed and connected. As the location of the pins is identical in the x-axis in the top- and bottom-partition cells, the routing tool creates long vertical wires crossing two partition cells. These additional 3 $\sim$ 4 metal layers used to connect the pins of the top- and bottom-partitioning cells are called "dummy wires" because their only function is to get logical connection between the two tiers in the physical design. The delay and RC parasitics associated with these wires will not be considered in the final M3D IC.

In an M3D IC, the topmost metal layer of the bottom tier is connected to the bottommost metal layer of the top tier using an MIV. To emulate this connectivity in a 2D IC where the top and bottom tiers are placed adjacent to each other, a mechanism to connect the bottommost metal layer (i.e., M1 in Fig. 8) in the top-partition view with the topmost metal layer (i.e., M6 in Fig. 8) in the bottom-partition view is required. This is achieved through "anchor cells." An anchor cell is a dummy cell which implements buffer logic. Anchor cells model zero-delay virtual connection between a dummy wire and one of the metal layers. After connecting

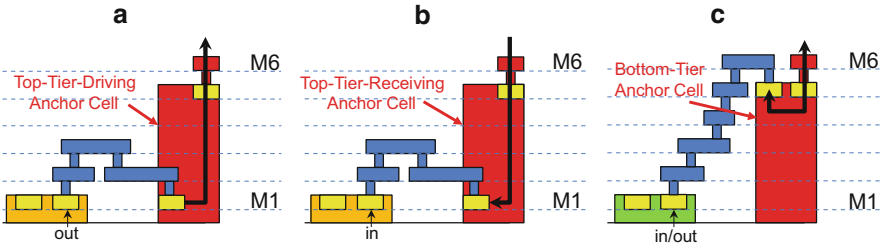


Fig. 13 Three types of anchor cells: (**a**) a top-tier-driving anchor cell, (**b**) a top-tier-receiving anchor cell, and (**c**) a bottom-tier anchor cell

the two partition cells with dummy wires, anchor cells are placed below the pins in each partition view. In this step, only anchor cells are placed but not logic cells.

Depending on the partition using anchor cells and metal layer to which a dummy wire needs to be virtually connected, three flavors of anchor cells exist: (1) top-tier-driving anchor cells (Fig. 13a), which are placed in the top partition, receiving signals from M1 of top partition, and driving a dummy wire, (2) top-tier-receiving anchor cells (Fig. 13b), which send signal in the reverse direction, and (3) bottom-tier anchor cells (Fig. 13c), which are placed in the bottom partition, connecting a dummy wire to top metal layer of the bottom partition. After placement, anchor cells and the corresponding MIV ports are connected.

Next, all hierarchies are flattened, i.e., the top view and both partition views are assembled projecting all anchor cells in two partition views and dummy wires in the top view into a single design. Figure 12b shows the assembled design.

With the assembled design, the delay of dummy wires is set to zero, and anchor cells and dummy wires are set to be fixed, so that their location cannot be modified. These sets of anchor cells and dummy wires effectively act as "wormholes" which connect the bottommost metal layer of the top partition and the topmost metal layer of the bottom partition without delay emulating the behavior of MIVs (the MIV RC parasitics are added in the final timing stage).

Then regular 2D IC design flow is performed, which involves all the design stages including placement, post-placement optimization, clock tree synthesis (CTS), post-CTS optimization, routing, and post-route optimization. Owing to (1) "wormholes," which provide virtual connection between the bottommost metal layer of the top partition and the topmost metal layer of the bottom partition, and (2) the hard fence, which sets the boundary for top and bottom partition, the tool places each tier in its separate 2D partitioned space with virtual connections between them, resulting in a cascade-2D design.

CTS in cascade-2D design flow is performed as regular 2D IC design flow. A clock signal is first divided into two branches in the top partition. One of the branches is used for generating the clock tree in the top partition, and the other branch is connected to the bottom partition through a set of anchor cells and a dummy wire and used for generating the clock tree in the bottom partition.

Figure 12c shows the resulting cascade-2D design. Although the delay of dummy wires is set to zero, their RC parasitics still exist in this stage of the design.

Therefore, the cascade-2D design is again split into top and bottom partitions, pushing all cells and wires to the corresponding partitions except dummy wires. Then, RC parasitics for each partition are extracted. The final M3D IC is created by connecting these two extracted designs with MIV RC parasitics. Timing and power analysis is done on the final M3D IC.

Impact of New Monolithic 3D IC Design Flow

Same experimental setup as described in section Implementation Methodology is used to gauge the benefits of cascade-2D design flow over shrunk-2D design flow.

Power and Performance Benefit

Figure 14 shows the GDS layouts of 2D and cascade-2D M3D ICs of the commercial, in-order, 32-bit application processor on target frequency of 1.0 GHz in 28 nm, 14/16 nm, as well as 7 nm technology nodes.

Timing analysis of the 2D IC indicates functional modules A and B in Fig. 15a have a large number of timing paths crossing them. In the cascade-2D M3D IC, those modules are floorplanned on top of each other minimizing the distance between them using MIVs, whereas those functional modules are floorplanned side by side in the 2D IC. This vertical integration reduces the wire-length of signals crossing the modules as well as the standard cell area of the modules because of reduced wire RC parasitics.

The normalized total power consumption of the 2D and cascade-2D M3D ICs across technology nodes is shown in Fig. 16. Cascade-2D M3D ICs consume less power in all cases. Hence, at iso-power, M3D ICs run at higher frequencies compared to the 2D ICs. For example, in the 14/16 nm technology node, M3D ICs can have 25% higher performance at the same total power compared to the 2D ICs. Figure 17 shows power saving comparison between cascade-2D M3D and shrunk-2D M3D ICs from their 2D counterparts. Cascade-2D M3D ICs show up to $3 \sim 4\times$ better power saving than shrunk-2D M3D ICs depending on the technology

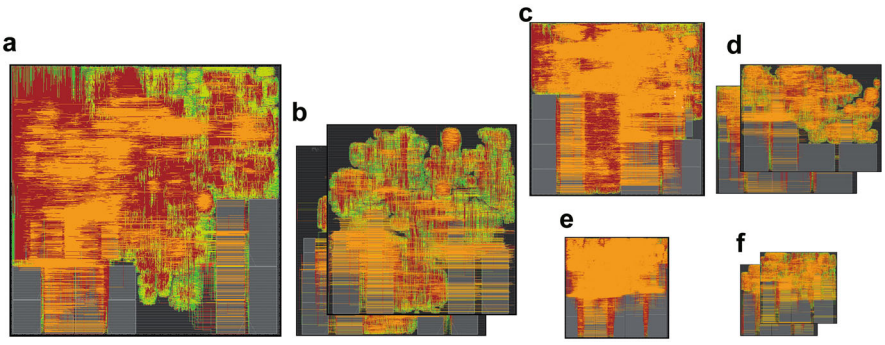


Fig. 14 GDS layouts of (**a**) 28 nm 2D, (**b**) 28 nm cascade-2D M3D, (**c**) 14/16 nm 2D, (**d**) 14/16 nm cascade-2D M3D, (**e**) 7 nm 2D, and (**f**) 7 nm cascade-2D M3D ICs of the application processor at 1.0 GHz

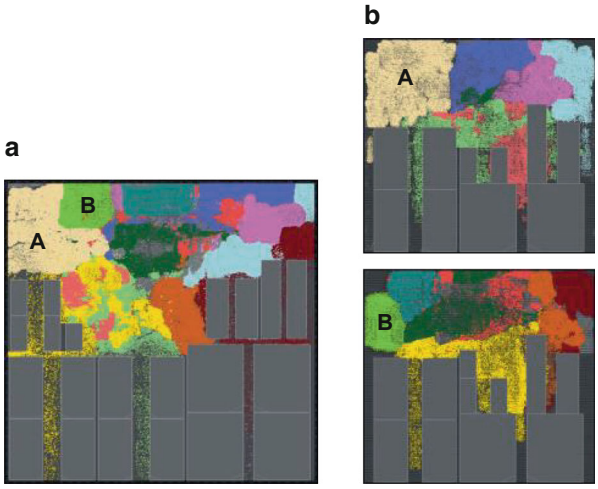
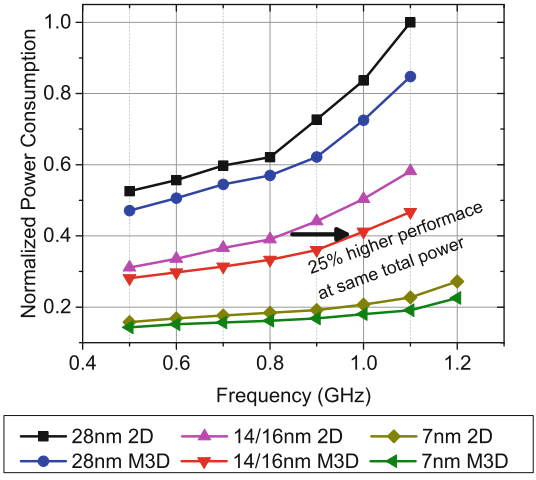


Fig. 15 Color map of functional modules in application processor 7 nm (**a**) 2D IC and (**b**) cascade-2D M3D IC of commercial application processor at 1.0 GHz

Fig. 16 Normalized power consumption of 2D and cascade-2D M3D ICs in 28 nm, 14/16 nm, and 7 nm technology nodes



node and design frequency. In the best-case scenario, M3D IC shows 20% power reduction than the 2D IC (14/16 nm technology node at 1.1 GHz frequency) at the same performance point.

Comparison to Shrunk-2D Design Flow

The primary advantage of shrunk-2D M3D ICs comes from reduced wire-length, which results in reduced wire capacitance switching power dissipation. As shown in Fig. 18, shrunk-2D M3D ICs reduce wire-length by 20 $\sim$ 25% consistently across technology nodes and frequencies. Wire-length reduction is mainly attributed to

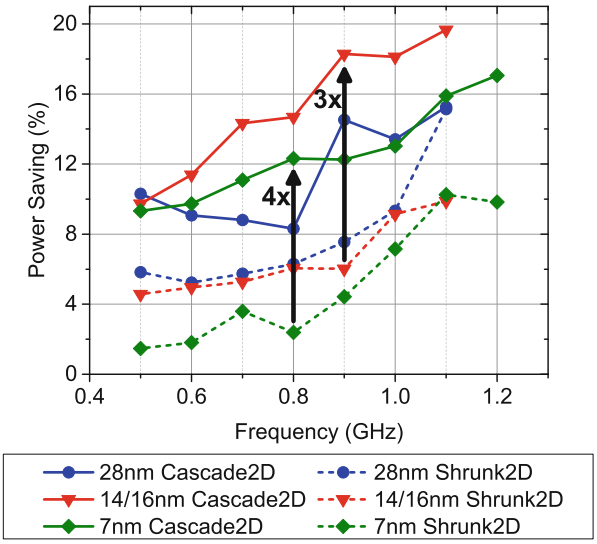


Fig. 17 Power saving of cascade-2D M3D (solid lines) and shrunk-2D M3D (dotted lines) ICs over 2D ICs in 28 nm, 14/16 nm, and 7 nm technology nodes

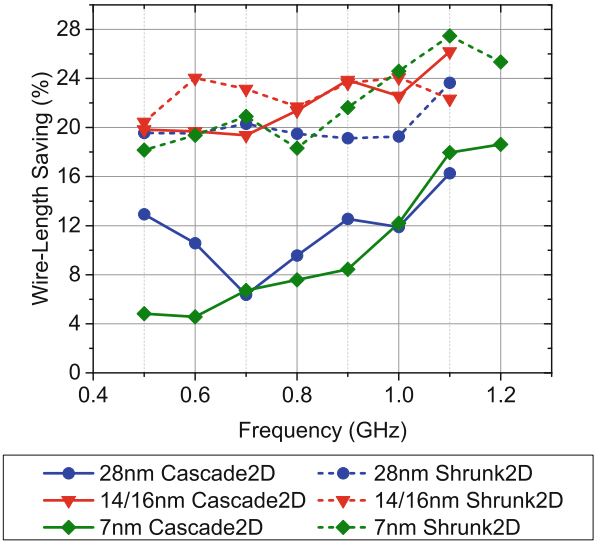


Fig. 18 Wire-length reduction comparison between cascade-2D (*solid lines*) and shrunk-2D (*dotted lines*) M3D ICs over 2D ICs

vertical integration between cells through MIVs. Table 4 compares the number of MIVs shrunk-2D M3D and cascade-2D M3D ICs. Since shrunk-2D design flow partitions cells into two tiers, whereas cascade-2D design flow partitions functional blocks, the number of MIVs in shrunk-2D M3D ICs is an order of magnitude higher

Table 4 Number of MIVs in application processor M3D ICs in 28 nm, 14/16 nm, and 7 nm technology nodes

MIV count	28 nm	14/16 nm	7 nm
Cascade-2D M3D IC	7545	7545	7545
Shrunk-2D M3D IC	164,553	120,770	99,587

Table 5 Normalized iso-performance comparison of 2D, shrunk-2D M3D, and cascade-2D M3D ICs with application processor in 28 nm, 14/16 nm, and 7 nm technology nodes. All values are normalized to corresponding 28 nm 2D parameters. Capacitance and power values are normalized to 28 nm 2D total capacitance and 28 nm 2D total power, respectively

Parameters	28 nm	14/16 nm	7 nm	28 nm	14/16 nm	7 nm	28 nm	14/16 nm	7 nm
Std. cell area	1	0.331	0.077	−7.6%	−6.8%	−7.5%	−9.5%	−11.9%	−8.8%
Wire-length	1	0.728	0.404	−19.3%	−24.1%	−24.6%	−11.9%	−22.6%	−12.2%
Wire cap	0.531	0.375	0.205	−18.1%	−14.2%	−13.7%	−9.5%	−19.7%	−19.2%
Pin cap	0.469	0.422	0.203	−12.1%	−6.3%	−9.7%	−11.1%	−13.2%	−7.9%
Total cap	1	0.797	0.408	−15.5%	−10.1%	−11.7%	−9.6%	−15.2%	−12.9%
Internal power	0.428	0.282	0.128	−4.8%	−7.6%	−4.7%	−14.5%	−15.2%	−11.1%
Net switching power	0.505	0.318	0.119	−13.4%	−10.6%	−10.1%	−13.0%	−20.8%	−15.1%
Leakage power	0.066	0.002	0.000	−7.7%	−4.0%	−2.0%	−9.5%	−7.7%	−2.8%
Total power	1	0.602	0.247	−9.3%	−9.1%	−7.2%	−13.4%	−18.1%	−13%

than that in cascade-2D M3D ICs. Better wire-length savings using shrunk-2D design flow can be attributed to the large number of MIVs.

The large number of MIVs in shrunk-2D M3D ICs helps to reduce wire-length, but it also increases the total capacitance of MIVs, limiting the wire capacitance reduction. As shown in Table 5, although shrunk-2D M3D ICs reduce more wire-length than cascade2D M3D ICs in 14/16 nm and 7 nm designs, the wire capacitance reduction of cascade-2D M3D ICs higher than shrunk-2D M3D ICs. Additionally, there is a negative impact of the large number of MIVs on the wire capacitance mainly because of the bin-based partitioning scheme of shrunk-2D design flow (Panth et al. n.d.). While the bin-based partitioning helps distribute cells evenly on both tiers, it has a tendency to partition cells connected using local wires into two tiers, increasing the wire capacitance.

On the other hand, cascade-2D M3D ICs save their power mainly by reducing standard cell area. Shrunk-2D design flow uses a shrunk-2D design to estimate the wire-length and the wire RC parasitics of the resulting M3D IC. However, while shrinking process geometries, minimum width of each metal layer is also scaled, and extrapolation is performed by tools during RC extraction of wires. This extrapolation tends to overestimate wire RC parasitics, especially in scaled technology nodes, which results in a large number of buffers inserted in a design to meet timing (Chang et al. 2017). In cascade-2D design flow, buffers are inserted while implementing and optimizing top and bottom partitions simultaneously with

actual process geometries, cascade-2D design flow achieves more standard cell area than shrunk-2D design flow as shown in Fig. 19.

With a reduction in standard cell area, the cell density of the M3D IC reduces as well. Hence, leveraging this feature of M3D ICs to increase cell density and reduce die area, two separate M3D ICs are implemented using cascade-2D design flow, one with the same total die area as the 2D IC and another with 10% reduced area. Table 6 shows that similar power savings can be achieved with a reduced die area M3D IC. The ability to get reduced die area makes M3D stacking technology extremely attractive for mainstream adoption because less area directly translates to reduced costs.

Standard cell area reduction affects both internal power, pin capacitance switching power reduction, whereas wire-length reduction reduces only wire capacitance switching power. Figure 20 shows the power breakdown of 2D, cascade2D M3D, and shrunk-2D M3D ICs. As shown in the figure, the internal power and pin capacitance switching power, which depends on the standard cell area, account for

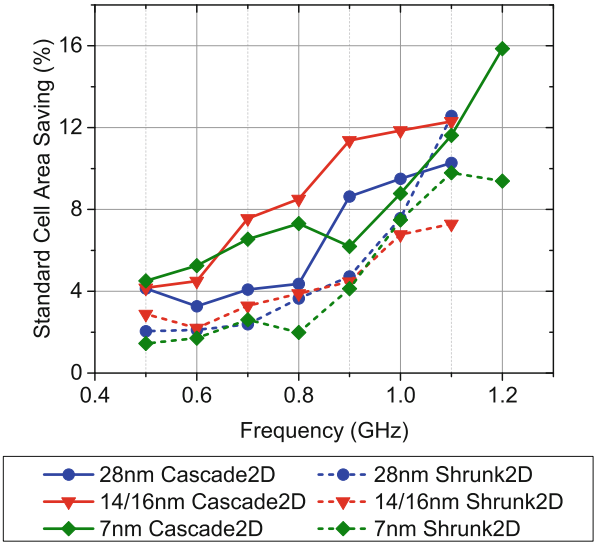


Fig. 19 Standard cell area saving in cascade-2D (*solid lines*) and shrunk-2D (*dotted lines*) M3D ICs over 2D ICs

Table 6 Normalized iso-performance power metric comparison of 2D and cascade-2D M3D IC with the same and 0.9× footprint at 1.1 GHz in 7 nm technology node

Parameters	2D	Cascade-2D M3D	
Die area	1	1	0.9
Density	69.7%	63.2%	71.1%
Total power	1	0.841	0.871

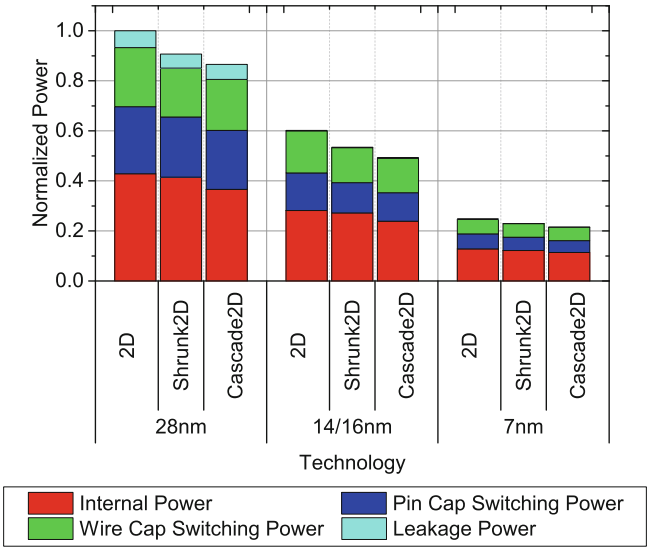


Fig. 20 Breakdown of the power consumption of 2D, shrunk-2D, and cascade-2D M3D ICs in 28 nm, 14/16 nm, and 7 nm technology nodes at 1.0GHz in foundry 28 nm, 14/16 nm, and 7 nm technology nodes

over 70% of the total power, and they contribute even more in 14/16 nm and 7 nm designs. Cascade-2D M3D ICs reduce more standard cell area compared to shrunk-2D M3D ICs by attacking 70% of the total power; they achieve better power savings consistently, even though the wire-length reduction of cascade-2D M3D ICs is less than shrunk-2D M3D ICs.

Power Supply Integrity of Monolithic Three-Dimensional Integrated Circuits

Motivation and Background

Challenges in designing a reliable PDN increase mainly due to lower supply voltage, faster operating clock frequency, and higher power density. Along with restricted budget of resources and cost, these challenges may cause functional failures and performance degradation due to parasitics-induced voltage drop in a non-ideal PDN. The total voltage drop is decomposed into a resistive component (IR-drop) and an inductive component ($L\mathrm{d}i/\mathrm{d}t$-drop). Increasing the metallization in a PDN can mitigate the resistive component of the voltage drop using wider interconnects while taking into account routing resources and cost budget.

Meanwhile, the inductance of a package including controlled collapsed chip connection (C4) bumps leads to significant $L\mathrm{d}i/\mathrm{d}t$-drop due to time-varying current drawn by cells in a die. In order to mitigate this drop, decoupling capacitors (decaps)

are utilized for local charge storage. Decaps can be placed on a die with decoupling cells (decap cells), or explicitly added in the package. However, this decap along with resistance and inductance of a PDN forms an RLC circuit resulting in its own resonance frequency (Larsson 1998). If the resonance frequency lies on the system's operating frequency range, a significant $L di/dt$-drop can be induced, and hence, it is crucial to have low input impedance across a wide range of frequencies.

While the PDNs of 2D ICs (i.e., 2D PDNs) have been explored actively (Larsson 1998) (Pant and Chiprout 2006), the PDNs of M3D ICs (i.e., M3D PDNs) have not been studied widely. A study for a system-level PDN for TSV-based 3D ICs is presented in Khan et al. (2011), but the PDNs in M3D ICs and in TSV-based 3D ICs show quite different characteristics due to their tier-connection method and achievable vertical integration density. In TSV-based 3D ICs, supply power is delivered directly to power pads of each tier through dedicated power TSVs, forming parallel resistive paths between multiple tiers. However, in M3D ICs, instead of having external power pads on the bottom tier, power MIVs are utilized to connect the bottommost metal layer of the top-tier PDN and the topmost metal layer of the bottom-tier PDN, consisting of series resistive paths across multiple tiers, which makes bottom-tier cells experience much longer resistive paths compared to TSV-based 3D ICs. Furthermore, irregular power MIV placement due to the cells on the top tier makes power delivery issue more complicated in M3D ICs. For these reasons, M3D ICs suffer much higher voltage drop in the static mode, especially on the bottom-tier cells as shown in Table 7 compared to TSV-based 3D ICs (Khan et al. 2011).

Although the series resistive paths of an M3D PDN worsen the voltage drop in the static mode, they benefit the voltage drop in the dynamic mode by improving resiliency against AC current noise, which will be discussed in later. Thus, the difference in the voltage drop between the 2D and the M3D IC in the dynamic mode is 7.3%, which is similar to TSV-based 3D ICs (Khan et al. 2011).

System-Level Power Delivery Network Analysis for Monolithic 3D ICs

The PDNs of 2D and M3D ICs are compared taking two analysis modes into account. The static mode is a vector-less analysis mode wherein the switching activity of cells is averaged into a single instance. In the dynamic mode, a real workload-based (i.e., vector-based) power analysis is performed for a given period of time. The dynamic mode thus incorporates the impact of inductive transients by taking into account workload-dependent time-varying current flow.

Table 7 Static and dynamic worst instance voltage drop in discrete cosine transform (DCT) 2D and M3D ICs

	2D	M3D	Δ%
Static voltage drop (mV)	27.6	68.1	146.7%
Dynamic voltage drop (mV)	323.4	346.9	7.3%
Δ%	1073.9%	507.1%	–

System-Level Power Delivery Network Modeling

In order to perform an in-depth PDN analysis, it is crucial to build a system-level PDN model. Figure 21 shows a simplified representation of an M3D PDN. It consists of a system model and a die model, and the system model is categorized into C4 bump, package, and printed circuit board (PCB) models (Pant and Chiprout 2006).

In Fig. 21, the resistance of the metal wire, $R_{PDN,eq}$, represents the equivalent resistive parasitics of the metal wires consisting the PDN, and the implicit decap of the die, $C_{PDN,eq}$, consists of the equivalent capacitance of the PDN metal wires, non-switching device capacitance, and coupling capacitance between N-well and substrate. The current drawn by switched cells is lumped and modeled as an AC load current source, $I_{LOAD,eq}$.

A representative lumped system model is created based on the parameters obtained from Pant and Chiprout (2006) and Das et al. (2015). The C4 bumps and power-line traces in the package and PCB are modeled as a series connection of a resistor and an inductor (C4 bumps: $R_{C4} = 1$ mΩ and $L_{C4} = 10$ pH; package: $R_{PKG} = 10$ mΩ and $L_{PKG} = 100$ pH; PCB: $R_{PCB} = 5$ mΩ and $L_{PCB} = 1$ μH), and a DC voltage source supplies power on the PCB. The inductor and the capacitor used in the voltage regulator module (VRM) LC-tank filter are incorporated within the PCB parasitics.

Since the implicit decap alone is not sufficient to keep the design in safe voltage drop region from Ldi/dt-drop, explicit decaps are deployed both on the die using decap cells, $C_{DIE_DC,eq}$, and on the package and PCB using discrete decaps, C_{PKG_DC} (= 400 nF) and C_{BULK_DC} (= 400 μF), respectively. The discrete decaps are modeled by a capacitor connected to an effective resistor and inductor in series ($R_{PKG_DC} = 20$ mΩ and $L_{PKG_DC} = 200$ pH; $R_{BULK_DC} = 10$ mΩ and $L_{BULK_DC} = 2$ nH). These explicit decaps and the implicit decap of the die act as charge storage elements and prevent system failure or performance degradation due to severe Ldi/dt-drop.

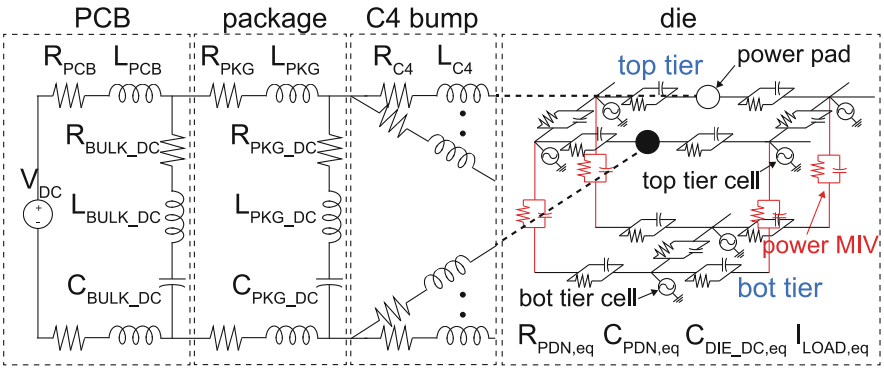


Fig. 21 Simplified model of a system-level M3D PDN structure

Analysis on Power Supply Integrity of Monolithic 3D ICs

Monolithic 3D IC Power Delivery Network Design Flow

In our design flow, we route power delivery network (PDN) before proceeding to signal routing in M3D designs. This is because signal and power/ground MIVs should not overlap with each other. In this case, we give a priory to PDN so that the IR-drop issue is addressed in early part of the design flow. Figure 22 shows our overall design flow, where the M3D PDN routing is done manually, while the subsequent signal routing is done with shrunk-2D design flow (Panth et al. n.d.).[3]

Once the top- and bottom-tier designs with PDNs are obtained for the final M3D IC, the designs are merged into a single M3D IC, and timing and power analysis as well as PDN analysis is performed.

Technology Nodes and Design Libraries

Three benchmarks, DCT, AES-128, and JPEG from OpenCores are used as benchmarks, and NanGate FreePDK45 Open Cell Library is used to synthesize, place, and route the designs. 2D ICs and each top and bottom tier of M3D ICs are implemented using seven metal layers. The footprint of the 2D ICs are determined such that the cell utilization is 60%, and the M3D ICs have half the footprint of the 2D counterparts. Target frequencies of each benchmark are fixed to their maximum operating frequency in the technology node.

Table 8 summarizes the resources used on each metal layer to build the 2D and M3D PDNs. The dimensions of power rails are determined targeting the maximum instance IR-drop to be 5% of nominal voltage (1.1 V) for the 2D ICs of all benchmarks in static rail analysis. For fair comparison, the same metrics are used for both the top and bottom tier of the M3D ICs. Power rails on M1 and M2 layers are tightly coupled and run in parallel in the horizontal direction of the designs, and

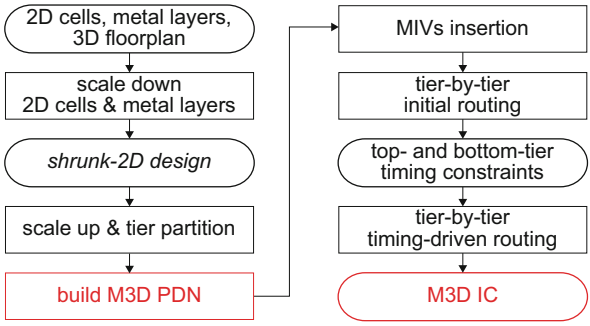


Fig. 22 Extended shrunk-2D design flow to insert a PDN

[3]We used shrunk-2D design flow in this study instead of other flows that are published in the literature. This is because shrunk-2D was the only flow that supported PDN routing at the time of this manuscript. However, our results should not depend heavily on which M3D signal routing to be used in the overall flow.

Table 8 Width, pitch, and utilization of the 2D and M3D PDNs. Same specs are used for both 2D and M3D (both top and bottom tier) ICs

Metal layer	Direction	Width (μm)	Pitch (μm)	Utilization
M2	H	0.07	1.4	10.0%
M5	V	0.28	14	20.6%
M6	H	0.28	14	20.6%
M7	V	0.8	42	11.1%

Table 9 Design metrics and decoupling capacitance of the created decap cells

Cell name	Cell width (μm)	Cell height (μm)	Capacitance (fF)
DECAP_ × 1	0.19	1.4	3.4
DECAP_ × 2	0.38	1.4	6.8
DECAP_ × 4	0.76	1.4	13.7
DECAP_ × 8	1.52	1.4	27.3
DECAP_ × 16	3.04	1.4	54.7
DECAP_ × 32	6.08	1.4	109.4

M2 and M5 power rails are connected with only via arrays, which cross M3 and M4. M5 to M7 power rails form a mesh structure to distribute power across the chip.

Since NanGate FreePDK45 Open Cell Library does not provide decap cells, decap cells are created with various sizes for the experiment. Table 9 shows the size and decoupling capacitance of the decap cells. The decoupling capacitance of each cell is derived using the method presented in Bozorgzadeh and Afzali-Kusha (2008). With the fully placed and routed 2D and M3D ICs, decap cells are first placed next to clock buffers driving the clock pins of flip-flops, which usually suffer from high $L\mathrm{d}i/\mathrm{d}t$-drop. Then, rest of decap cells are placed in empty area of the designs to meet a target decoupling capacitance of the chip.

The power and ground pads of the designs are located on the top metal layer of designs (M7 for the 2D ICs, M7 of the top tier for the M3D ICs) with 120 μm spacing, which model the C4 bumps of the designs.

Analysis Methods

Figure 23 shows the number of switched cells in the DCT design during a workload-based simulation. The vector-based power consumption in Table 10 is measured during the time step which shows the highest switching activity throughout the simulation (blue bar in Fig. 23), while the statistical power consumption of the designs is calculated assuming the switching ratio of the primary input and sequential logic as 20% and 10%, respectively. Therefore, the dynamic power (i.e., internal + net switching power) shows significant difference between two analysis methods, whereas the static power (i.e., leakage power) remains similar.

The M3D ICs offer power benefit over their 2D counterparts. Since M3D ICs utilize short vertical integration with MIVs instead of using long metal wires on the xy-plane, the wire-length of the designs is reduced as shown in Table 10, offering

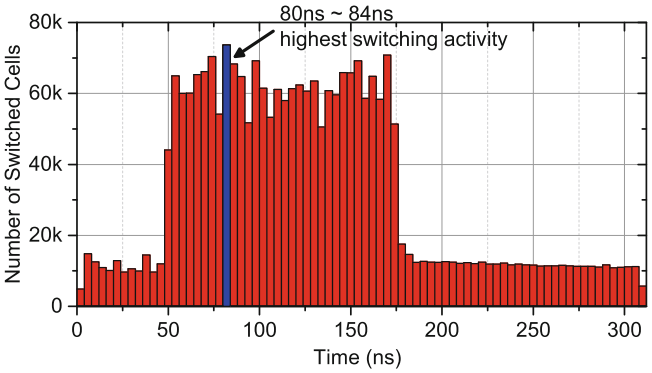


Fig. 23 Number of the switched cells in a DCT design during a workload-based simulation. Only the time period which shows the highest switching activity (blue bar) is used for the analysis

net switching power saving. In addition, since the cells drive the reduced wire-load, the number of buffers as well as the drive-strength of the cells decreases, which, in turn, reduces the standard cell area, hence, showing benefits on the internal and leakage power consumption.

Instance voltage drop is used, which is the voltage drop a cell experiences as Eq. 5.

$$\Delta V_{\text{inst}} = \left(V_{\text{DD,nom}} - V_{\text{DD,act}}\right) + \left(V_{\text{SS,act}} - 0\right), \tag{5}$$

where ΔV_{inst} is instance voltage drop and $V_{\text{DD,nom}}$ is nominal voltage. $V_{\text{DD,act}}$ and $V_{\text{SS,act}}$ are actual voltage level on the power and ground pin of the cell. Instance voltage drop can be further decomposed into the voltage drops on each metal layer. Figure 24 shows the decomposed voltage drop at each metal layer (voltage values in black) for the cell experiencing the worst instance voltage drop in static rail analysis of the JPEG M3D IC, showing how much IR-drop the power rails in each metal layer has contributed to the total instance IR-drop.

Static Rail Analysis

Since static rail analysis is based on statistical power consumption, which summarizes the behavior of designs, only IR-drop can be analyzed. Figure 25 shows the breakdown of the worst instance IR-drop into each metal layer consisting of the PDN of the 2D and M3D ICs. The M3D ICs show approximately 2× higher IR-drop on average compared to the 2D counterparts. Since M3D ICs utilize more metal layers for their PDN structure to deliver power to the bottom-tier cells, those cells experience worse IR-drop compared to the top-tier cells (the dashed box from M1B to M7B in Fig. 25).

Another reason for the higher IR-drop in M3D ICs is irregular placement of power MIVs which connect PDNs of two tiers. Figure 26 illustrates the impact of irregular power MIV placement. In M3D PDNs, current flowing in metal layers

Table 10 Iso-performance design and power metric comparison of 2D and M3D ICs used for static and dynamic rail analysis. Both statistical and vector-based power simulations are conducted. Δ% for M3D ICs is calculated with respect to the 2D counterparts

		DCT		AES-128		JPEG	
Benchmark		2D	M3D (Δ%)	2D	M3D (Δ%)	2D	M3D (Δ%)
Frequency (MHz)		500	500 (−)	1000	1000 (−)	500	500 (−)
Footprint (μm)		369 × 368	260 × 260 (−50.2%)	509 × 507	360 × 360 (−49.8%)	897 × 895	634 × 636 (−49.8%)
C4 bump count		9	4 (−55.6%)	16	9 (−43.8%)	64	25 (−60.9%)
Std. cell area (μm^2)		85,432	85,312 (−0.1%)	166,560	163,938 (−1.6%)	503,070	503,068 (0.0%)
Wire-length (mm)		784.0	723.1 (−7.8%)	1921.2	1708.3 (−11.1%)	3770.3	3730.1 (−1.1%)
Total cap. (pF)		236.1	220.0 (−6.8%)	568.6	500.3 (−12%)	1186.4	1153.7 (−2.8%)
Signal MIV count		–	11,753	–	50,589	–	58,807
Statistical Power Analysis	**Internal Power (mW)**	16.4	16.2 (−1.6%)	49.3	48.5 (−1.6%)	109.4	109.8 (0.4%)
	Net switching power (mW)	15.9	13.4 (−15.8%)	46.8	41.1 (−12.3%)	93.4	89.3 (−4.4%)
	Leakage Power (mW)	0.8	0.7 (−2.8%)	1.9	1.7 (−7.6%)	4.5	4.4 (−1.9%)
	Total power (mW)	33.1	30.3 (−8.4%)	98.0	91.3 (−6.8%)	207.3	203.6 (−1.8%)
Vector-based Power Analysis	**Internal Power (mW)**	48.0	51.2 (6.7%)	197.3	186.0 (−5.7%)	218.1	222.1 (1.8%)
	Net switching power (mW)	36.7	31.3 (−14.7%)	134.8	87.9 (−34.8%)	88.8	91.8 (3.4%)
	Leakage power (mW)	0.7	0.7 (−3.0%)	2.0	1.8 (−7.2%)	4.7	4.6 (−2.1%)
	Total power (mW)	85.4	83.2 (−2.6%)	334.0	275.7 (−17.5%)	311.6	318.5 (2.2%)

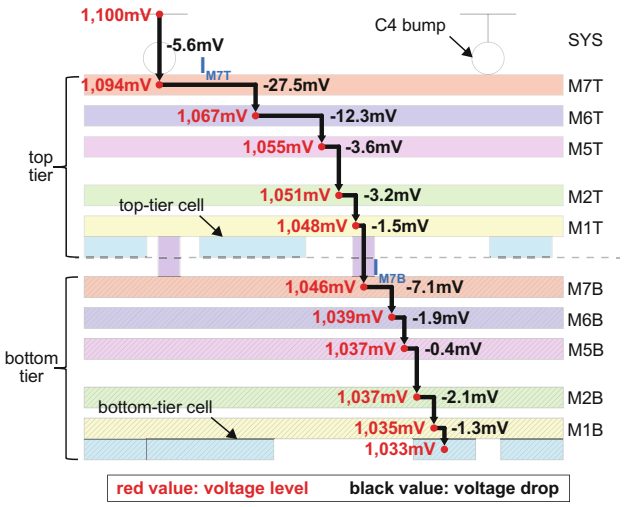


Fig. 24 Illustration describing how the worst instance IR-drop can be decomposed into each metal layer, showing voltage drops on the power rails in each metal layer (values in black) and voltage level on each metal layer along the IR-drop path (values in red)

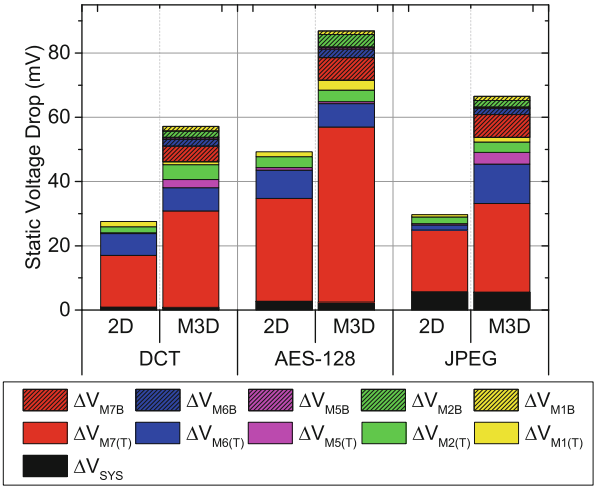


Fig. 25 Breakdown of the worst instance IR-drop across the metal layers comparing 2D and M3D ICs. M7B denotes M7 of the bottom tier in the M3D ICs. SYS represents the system model including C4 bump, package, and PCB model

of the top tier is greater than those of the bottom tier (e.g., $I_{M7_T} > I_{M7_B}$ in Fig. 26) since top-tier metal layers deliver current to both top- and bottom-tier cells, whereas only current drawn by bottom-tier cells flows on bottom-tier metal layers. Therefore, the minimum IR-drop path in Fig. 26 to deliver power to a bottom-tier cell utilizes

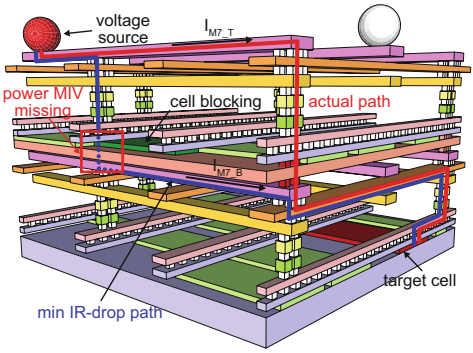


Fig. 26 Current path to deliver power to a target cell showing the impact of missing power MIVs. A top-tier cell is blocking a power MIV along the minimum IR-drop path, so the current is delivered through an alternative path

Table 11 Average amount of current flowing through C4 bumps in 2D and M3D ICs

Design	2D	M3D	Δ%
DCT	3.34 mA	6.89 mA	106.1%
AES-128	5.48 mA	9.23 mA	68.2%
JPEG	2.87 mA	7.34 mA	155.7%

the minimum length of top-tier power rails. However, the path can be blocked by a missing power MIV. The absence of power MIVs stems from top-tier cells since MIVs cannot penetrate those cells in order to preserve their active areas. In this case, the current needs to flow through an alternative path shown as actual path in Fig. 26, which utilizes longer top-tier metal wires and hence exhibits worse IR-drop due to higher current in those wires.

Reduced number of C4 bumps in M3D ICs also degrades voltage integrity. As the footprint of an M3D IC is half of its 2D counterpart, the number of the C4 bumps that can be placed on an M3D IC is approximately half of those in the 2D IC as shown in Table 10. This affects the amount of the current flowing through each C4 bump. Table 11 compares the current flowing through C4 bumps in the 2D and M3D ICs. Up to 155.7% higher current flows through the C4 bumps in the M3D ICs incurring significant difference in IR-drop on the top metal layers (i.e., M7 and M6 in Fig. 25).

Dynamic Rail Analysis

Unlike static rail analysis, dynamic voltage drop consists of two categories, IR-drop and $L di/dt$-drop. $L di/dt$-drop has significantly higher impact on the voltage drop since dynamic rail analysis is performed for two clock cycles with the maximum switching activity in a real workload. The voltage drop of the M3D ICs is 11.3% higher on average than the 2D ICs as shown in Fig. 27, which is much less than that in the static rail analysis.

First, in the dynamic rail analysis, the difference between the voltage drop of the metal layers in the 2D ICs and the metal layers of the top-tier PDN of M3D ICs is significantly less than that in the static rail analysis. The reduced voltage drop on those metal layers first results from 3D placement of decaps in M3D ICs.

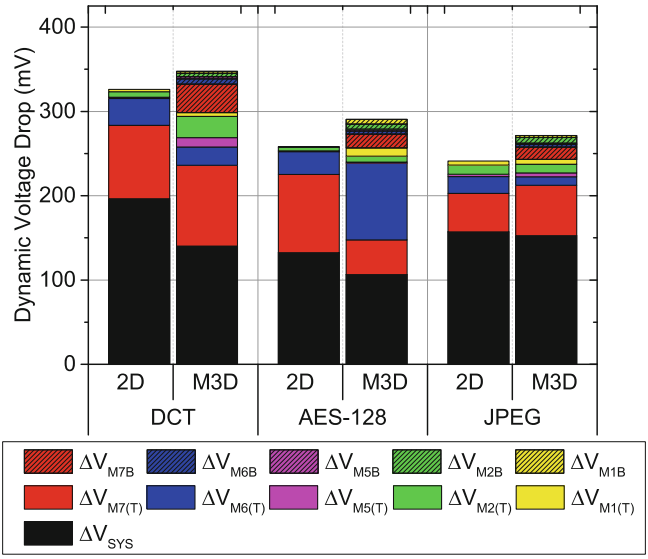


Fig. 27 Breakdown of the worst instance dynamic voltage drop (= IR-drop + *L*di/dt-drop) across the metal layers comparing 2D and M3D ICs

As discussed, decaps from non-switching devices (implicit) and decap cells (explicit) in a design act as charge reservoir, preventing nearby cells from experiencing sudden high *L*di/d*t*-drop. In M3D ICs, *L*di/d*t*-drop is reduced because of decap cells in both, the xy-plane, as in the case of 2D ICs, as well as decap cells in the adjacent tier (the z-axis), as in TSV-based 3D ICs (Khan et al. 2011).

Figure 28 shows the maximum voltage drop experienced at the C4 bumps comparing the DCT 2D and M3D ICs with and without decap cells. The decoupling capacitance of the 2D and M3D ICs with decap cells is targeted to 30% of their total capacitance. Even though the decoupling capacitance added to the M3D IC (= 72.6 pF) is smaller than the 2D IC (= 77.9 pF) due to the lower total capacitance of the DCT M3D IC, it benefits more from the added decap cells than the 2D IC as it utilizes decaps in the z-axis as well. Another reason for the smaller gap between the 2D and M3D IC voltage drop in the dynamic rail analysis is the reduced voltage drop on the system model as shown in Fig. 27 due to the varying impedance seen from the die depending on operating frequency, which will be discussed in the next sub-section.

Frequency- and Time-Domain Analysis

As shown in Fig. 21, implicit and explicit decaps on a die model are coupled with inductors in a system model, forming an RLC circuit. The RLC circuit has its own resonance frequency, causing significant voltage drop on the PDN even with small changes in load current. Explicit decaps on the package and PCB also form

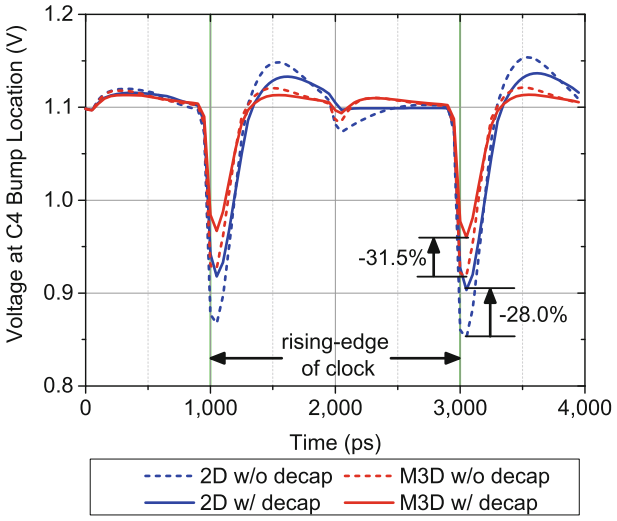


Fig. 28 Comparison of the worst voltage drop experienced at C4 bumps showing the impact of decoupling capacitance in 2D and M3D ICs. The decoupling capacitance is set to 30% of the total capacitance of each design

Table 12 Effective resistance and capacitance of 2D and M3D PDNs. Resistance is in Ω, and capacitance is in nF

Parameter		2D	M3D	Δ%
DCT	$R_{PDN,eq}$	0.221	0.812	267.5%
	$C_{PDN,eq} + C_{DIE_DC,eq}$	0.232	0.211	−8.7%
	Product of *R* and *C*	0.051	0.172	235.5%
AES-128	$R_{PDN,eq}$	0.164	0.341	108.1%
	$C_{PDN,eq} + C_{DIE_DC,eq}$	0.528	0.439	−16.8%
	Product of *R* and *C*	0.086	0.150	73.1%
JPEG	$R_{PDN,eq}$	0.076	0.202	164.4%
	$C_{PDN,eq} + C_{DIE_DC,eq}$	1.290	1.220	−5.4%
	Product of *R* and *C*	0.098	0.246	150.1%
Average product of *R* and *C*		0.079	0.189	140.4%

RLC circuits with the corresponding inductors, showing their unique resonance frequencies.

To perform an in-depth frequency- and time-domain analysis on a PDN, a reasonable die model which represents 2D and M3D full-chip System-on-Chip (SoC) is needed. Since the benchmarks used in this work are small compared to full-chip SoCs, their parameters are used to create a full-chip die model. Table 12 shows the effective resistance and capacitance of the PDN of each benchmark ($R_{PDN,eq}$ and $C_{PDN,eq} + C_{DIE_DC,eq}$ in Fig. 21, respectively). As a design becomes larger, the capacitance of its PDN increases due to the increased ground and coupling capacitance of the PDN, while the resistance becomes smaller because more number of parallel resistive paths to the cells are available. To ease in modeling, the average

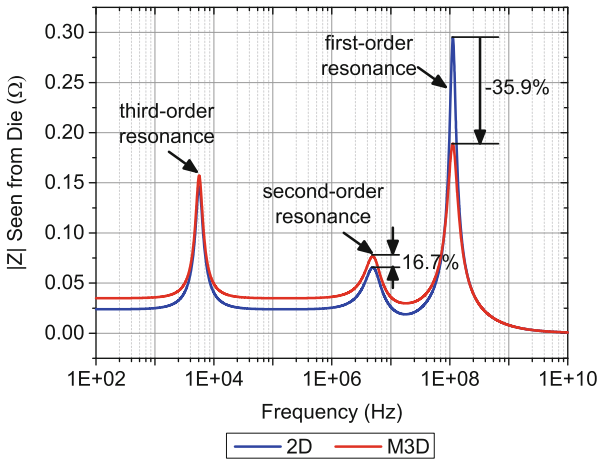


Fig. 29 Impedance seen from the die by sweeping the frequency of AC load current source, $I_{\mathrm{LOAD,eq}}$

of the RC product from the three benchmarks is used, and a full-chip die is modeled by assuming $C_{\mathrm{PDN,eq}} + C_{\mathrm{DIE_DC,eq}} = 10$ nF, resulting in the associated resistances as 7.87 mΩ and 18.9 mΩ for the 2D and M3D ICs, respectively.

Figure 29 shows the frequency response of the 2D and M3D full-chip SoC sweeping the frequency of the AC load current source, $I_{\mathrm{LOAD,eq}}$. Three resonance frequency points are observed, first-order resonance caused by $C_{\mathrm{PDN,eq}} + C_{\mathrm{DIE_DC,eq}}$ coupled with L_{C4}, second-order resonance by $C_{\mathrm{PKG_DC}}$ with L_{PKG}, and third-order resonance by $C_{\mathrm{BULK_DC}}$ with L_{PCB}. While third-order and second-order resonance occurs at a few kHz and MHz range, the largest resonance, first-order resonance is in the range between 50 MHz and 200 MHz. Although the M3D IC shows 16.7% increase at second-order resonance frequency, as the operating frequencies of full-chip SoC at advanced technology nodes are in the range of first-order resonance frequencies, it is crucial to minimize the first-order resonance impact for a robust PDN.

As shown in the figure, the M3D IC exhibits 35.9% lower peak impedance at first-order resonance frequency because of high effective resistance of the M3D PDN due to series resistive paths across tiers. An interesting point is that the high resistance of M3D PDNs, which worsens IR-drop, in fact, improves the resiliency against AC current noise by damping noise at worst-case resonance oscillation.

Figure 30a and b shows the improved resiliency of the M3D PDN, showing the time-domain response for a unit step, which models in-rush current simulation, and for a 112 MHz (first-order resonance frequency) unit sine-wave load current source. Equation 6 explains the die voltage response affected by first-order resonance for a unit step load current source (Pant and Chiprout 2006):

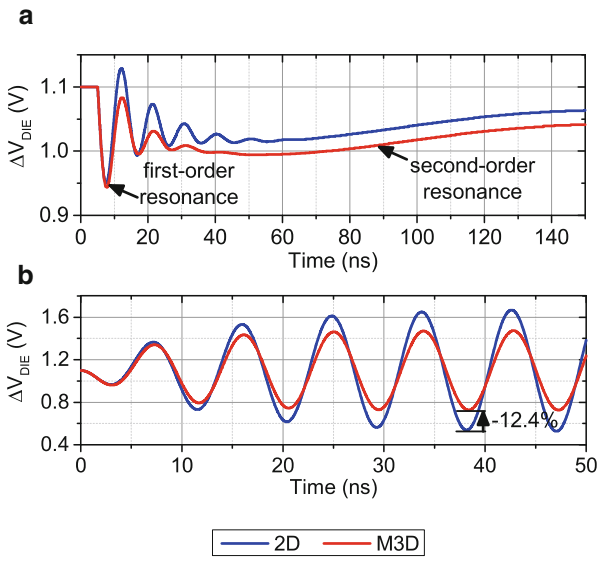


Fig. 30 Transient voltage response for (**a**) a unit step and (**b**) a unit 112 MHz (first-order resonance frequency) sine-wave load current source, $I_{\mathrm{LOAD,eq}}$. Third-order resonance frequency is not shown in (**a**) for brevity

$$\Delta V_{\mathrm{DIE}} \cong 2R + \sqrt{\frac{2L_{\mathrm{C4}}}{C_{\mathrm{DIE,eq}}}} \mathrm{e}^{-\frac{R}{2L_{\mathrm{C4}}}t} \sin\left(\omega_{\mathrm{r}} - \theta\right), \tag{6}$$

where $R = R_{\mathrm{PCB}} + R_{\mathrm{PKG}} + R_{\mathrm{C4}} + R_{\mathrm{PDN,eq}}$, ω_{r} and θ are first-order resonance frequency and phase, respectively. While the increased R in an M3D PDN worsens the IR-drop at cell (the first term in Eq. 6), it helps to reduce the second term, $L\mathrm{d}i/\mathrm{d}t$-drop. The improved resiliency for first-order resonance helps neutralize the voltage drop gap induced by second-order resonance in the worst voltage drop as shown in Fig. 30a and shows 12.4% less voltage drop with current source oscillating at first-order resonance frequency as shown in Fig. 30b.

Monolithic 3D ICs for Deep Neural Network Hardware

Motivation and Background

DNNs have become ubiquitous in many machine learning applications, from speech recognition (Deng et al. 2013; Graves et al. 2013) and natural language processing (Conneau et al. 2017), to image recognition (Krizhevsky et al. 2012; He et al. 2015) and computer vision (Karpathy and Fei-Fei 2017). Large neural network models have proven to be very powerful in all the stated cases, but implementing

high-speed (i.e., high-performance), energy-efficient DNN application-specific integrated circuit (ASIC) is still challenging because (1) the required computations consume large amounts of processing time and energy, (2) the memory needed to store the weights are prohibitive, and (3) excessive wire overhead exists due to a large number of connections between neurons, which makes a DNN ASIC a heavily wire-dominated circuit.

Modern DNNs may require >100 M parameters (Xiong et al. 2016) for large-scale speech recognition tasks. This is impractical using only on-chip memory due to power density and temperature instability (Liao et al. 2005), and hence offloading storage to an external DRAM is required. With the introduction of an external DRAM, however, the bottleneck for computation efficiency is now determined by the parameter fetching from DRAM (Sze et al. 2017). To mitigate this bottleneck, recent works have compressed the neural network weights "in architectural perspective" and substantially reduced the amount of computation required to obtain the final output (He et al. 2014; Han et al. 2015; Kadetotad et al. 2016; Cheng et al. 2017), which becomes crucial for efficient DNN ASICs. An alternate method of reducing the complexity caused by the vast requirement of memory for DNNs is in-training quantization of the network parameters (Courbariaux et al. 2015; Courbariaux et al. 2016). This method, however, is not explored.

With the weight-compressed DNN architecture, M3D stacking technology is adopted to further improve the energy-efficiency and performance "in physical design perspective."

Impact of Monolithic 3D ICs on On-Chip Deep Neural Networks Targeting Speech Recognition

In this chapter, the impact of M3D stacking technology on power, performance, and area is investigated with speech recognition DNN architectures that exhibit coarse-grain sparsity. M3D ICs reduce the total power consumption more effectively with compute-intensive workloads, compared to memory-intensive workloads. By placing memory blocks evenly on both tiers, M3D ICs reduce the total power consumption up to 22.3%. In addition, owing to the reduced footprint and vertical integration, M3D ICs offer performance improvement over 2D ICs, especially in architecture with complex combinational logics.

Deep Neural Network for Speech Recognition

In this section, topology, the training, and classification strategy are presented, which are used for the DNN architectures for speech recognition. In addition, the coarse-grain sparsification (CGS) is introduced, which effectively reduces area and computation overhead of DNNs.

DNN Topology

Starting from a fully connected DNN, a Gaussian Mixture Model (GMM) is adopted for acoustic modeling (Su et al. 2010). Since it has been shown that DNNs in

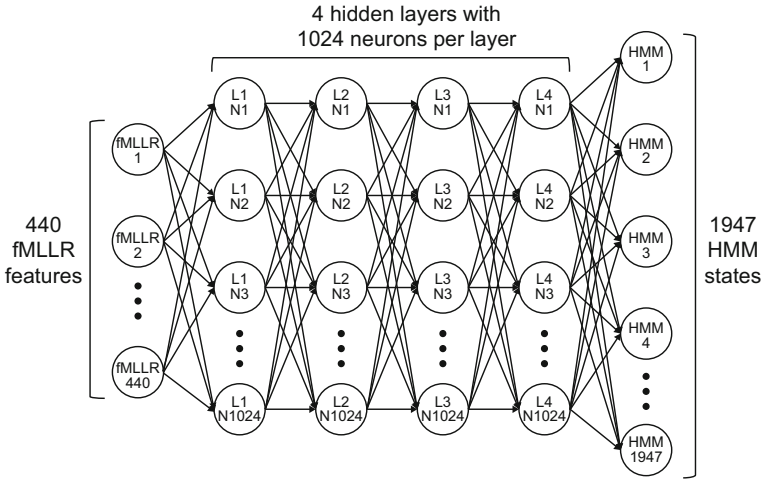


Fig. 31 Diagram of the DNN for speech recognition

conjunction with Hidden Markov Models (HMMs) increase recognition accuracy (Deng et al. 2013), a HMM is also employed to model the sequence of phonemes. The most likely sequence is determined by the HMM utilizing the Viterbi algorithm for decoding. Then, the CGS methodology presented in Kadetotad et al. (2016) is adopted in the DNN architecture to reduce the memory footprint as well as the computation for DNN classification.

As shown in Fig. 31, the DNN for speech recognition consists of 4 hidden layers with 1024 neurons per layer. There are 440 input nodes corresponding to 11 frames (5 previous, 5 future, and 1 current) with 40 feature-space Maximum Likelihood Linear Regression (fMLLR) features per frame. The output layer consists of 1947 probability estimates, and they are sent to the HMM unit to determine the best sequence of phoneme using the TIMIT database (Garofolo et al. 1993). The Kaldi toolkit (Povey et al. 2011) is utilized for the transcription of the words and sentences for the particular set of phonemes.

Deep Neural Network Training and Classification

The DNN is trained with the objective function that minimizes the cross-entropy error of the outputs of the network as Eq. 7.

$$E = -\sum_{\mathrm{i}=1}^{N} t_{\mathrm{i}} \cdot \ln\left(y_{\mathrm{i}}\right), \tag{7}$$

where N is the size of the output layer, y_{i} is the ith output node, and t_{i} is the ith target value or label. The mini-batch stochastic gradient method (Gardner 1984) is used to update the weights. The weight W_{ij} is updated in the $(k+1)^{\mathrm{th}}$ iteration using Eq. 8.

$$\left(W_{ij}\right)_{k+1} = \left(W_{ij}\right)_{k} + C_{ij}\Big(- lr\left(\Delta W_{ij}\right)_{k} + m\left(\Delta W_{ij}\right)_{k-1}, \tag{8}$$

where m is the momentum, lr is the learning rate, and C_{ij} is the binary connection coefficient between two subsequent neural network layers for CGS. In CGS, only the weights that correspond to the location where $C_{ij} = 1$ are updated. The change in weight for each iteration is the differential of the cost function with respect to the weight value:

$$\Delta W = \frac{\delta E}{\delta W} \tag{9}$$

such that the loss reduces in each iteration. The training procedure is performed on a graphics processing unit (GPU) with 32-bit floating point values.

After training, feed-forward computation is performed for classification, through matrix vector multiplication of weight matrices and neuron vectors in each layer to obtain the output of the final layer. The Rectified Linear Unit (ReLU) function (Krizhevsky et al. 2012) is used for the non-linear activation function at the end of each hidden layer.

Coarse-Grain Sparsification

To efficiently map sparse weight matrices to memory arrays, CGS methodology (Kadetotad et al. 2016) is employed. In CGS, connections between two consecutive layers in a DNN are compressed in a block-wise manner. An example of block-wise weight compression is demonstrated in Fig. 32. For a given block size of 16 × 16, it reduces a 1024 × 1024 weight matrix to 64 × 64 weight blocks. With a compression ratio of 87.5%, only eight weight blocks (= 12.5%) remain non-

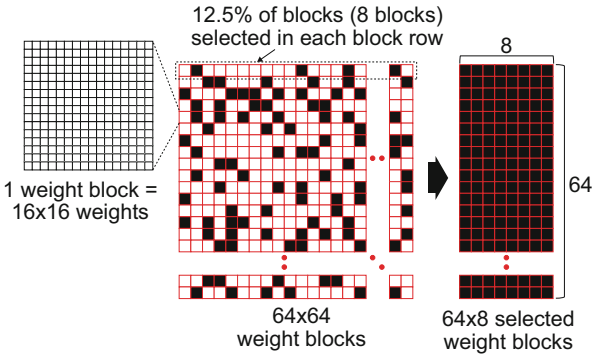


Fig. 32 An example of block-wise weight compression in CGS. A 1024 × 1024 weight matrix is divided into 64 × 64 weight blocks with each weight block having 16 × 16 weights (i.e., block size of 16 × 16). A total of 87.5% of weight blocks are dropped using CGS. The remaining 12.5% weight blocks are stored in memory

zero for each block row, thus allowing for efficient compression of the entire weight matrix with minimal index.

CGS, when compared to recent neural network compression algorithms such as in Han et al. (2016) and Cheng et al. (2015), offers simpler hardware implementation through CGS multiplexers and multiplier-accumulators (MACs). In Han et al. (2016), a complex sparse matrix vector multiplication module is required. On the other hand, the methodology in Cheng et al. (2015) offers to reduce the order of computations needed for a matrix of size n to O(nlogn) and reduce the space required to store the matrix to O(n). However, there is considerable loss in accuracy when the size of the matrix increases, and hardware for computing FFT and inverse fast Fourier transform (IFFT) is required. The issue of matrix size is resolved in Liao et al. (2017) using block-circulant matrices, but the advantage of using FFT and IFFT to compute matrix vector multiplications is lost if the size of the blocks reduces significantly. This restriction is not present if CGS is used.

GPU-accelerated DNN computations can also benefit from CGS. With CGS, along with the testing inference, training complexity can also be reduced due to the sparse nature of the weight matrices. The structured sparseness allows for writing customized GPU kernels that only need to operate on the non-zero elements, significantly speeding up training and reducing GPU power consumption as shown in (Gray et al. 2017).

In order to study the impact of M3D ICs on the power, performance, and area of different DNN architectures, the block sizes are swept for the compression ratio of 87.5%, and the two DNN architectures that have the two lowest phoneme error rates (PER) for the TIMIT dataset are selected for hardware implementation. The two architectures chosen are the DNN with 16 × 16 block size (DNN CGS-16) and the DNN with 64 × 64 block size (DNN CGS-64), as shown in Table 13.

Deep Neural Network Architecture Description

The block diagram of the CGS-based DNN architecture is shown in Fig. 33. The DNN operates on one layer at a time and consists of 16 MAC units that operate in parallel. The weights of the network are stored in the SRAM banks, while the input and output neurons are stored in registers. The finite state machine (FSM) coordinates the data flow such as layer control and computational resource allocation (i.e., MAC units).

Since the target compression ratio of the architectures is 87.5%, the neuron select unit chooses 128 neurons (12.5%) among 1024 input neurons that proceed to the MAC units. This selection-based computation eliminates unnecessary MAC

Table 13 Key parameters of the two CGS-based DNN architectures used in this work: DNN CGS-16 and DNN CGS-64

Parameter	DNN CGS-16	DNN CGS-64
Block size	16 × 16	64 × 64
Compression rate	87.5%	87.5%
Phoneme error rate	19.8%	19.9%

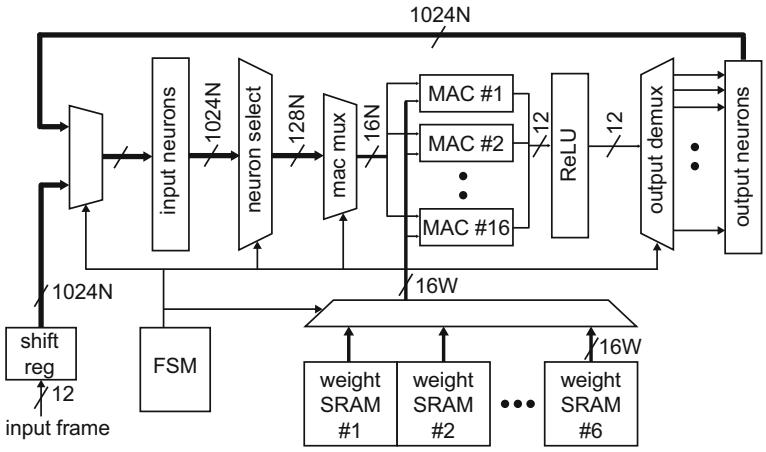


Fig. 33 Block diagram of the CGS-based DNN architecture for speech recognition

operations (i.e., MAC operation of neurons corresponding to zero weights in CGS-based weight matrix). The neuron select unit is controlled by the binary connection coefficients, and the coefficients are stored in the dedicated register file in the FSM unit.

The size of the register file is determined by the block size used in the DNN architecture. For example, for each hidden layer, eight weight blocks per each row of 64 × 64 weight blocks are selected for MAC operation in the DNN CGS-16 architecture (Fig. 32). Thus, eight multiplexers are required in the neuron select unit, and each multiplexer selects one weight block among 64 in a block row, so that each multiplexer requires six selection bits (=$\log_2 64$). Since there are 64 total block rows in the architecture, the total number of bits to obtain 64 × 8 selected weight blocks for a hidden layer is 3072 bits (= eight multiplexers × 6 selection bits × 64 block rows). Although the DNN has four hidden layers, the number of coefficients for the last hidden layer should be doubled because the number of neurons in the output layer (1947 HMM states) is almost 2× of other layers. Therefore, the size of the coefficient register file in the DNN CGS-16 is 15,360 bits (= 3072 bits × 5 effective layers). This value is calculated in the same way for the DNN CGS-64 architecture, resulting in 640 bits in total.

On-chip SRAM arrays store the compressed weight parameters in six banks for the four hidden layers and the output layer (∼2× parameters). The size of the SRAM bank is determined by the number of MAC units in the architecture. Since the DNN architectures operate 16 units in parallel, the row size of each SRAM bank is 128 bits (= 16 MAC units × 8-bit weight precision). Since 8192 rows are assumed for each SRAM bank, the total size of the six SRAM banks in the DNN is 6 Mb (= 6 banks × 128 bits × 8192 rows). This compact memory size with the CGS methodology enables the DNN to store the compressed weight parameters on chip.

Impact of Monolithic 3D ICs on Energy-Efficiency of Deep Neural Network Hardware

To analyze the advantage of M3D stacking technology on energy-efficiency of different DNN architectures, two DNN architectures (DNN CGS-16 and CGS-64) are implemented using TSMC® 28 nm HPM technology with a target clock frequency of 400 MHz. The footprint of 2D ICs is set by targeting cell density of 65%. M3D ICs are implemented using shrunk-2D design flow (Panth et al. n.d.). The impact of memory tier partitioning scheme is examined by comparing two memory floorplan schemes for M3D ICs, one with memory blocks on both tiers (i.e., M3D-both) and the other with memory blocks on a single tier only (i.e., M3D-one). In the M3D-both designs, memory blocks are evenly split on the top and bottom tiers using similar floorplan for both tiers. On the other hand, in the M3D-one designs, all standard cells are placed on one tier, and only memory blocks exist on the other tier. Figure 34 shows the GDS layouts of the implemented 2D and M3D ICs.

Area, Wire-Length, and Capacitance Comparisons

Iso-performance comparison of several key metrics of the 2D and M3D ICs is presented in Table 14. The M3D-both designs achieve 50.1% footprint reduction

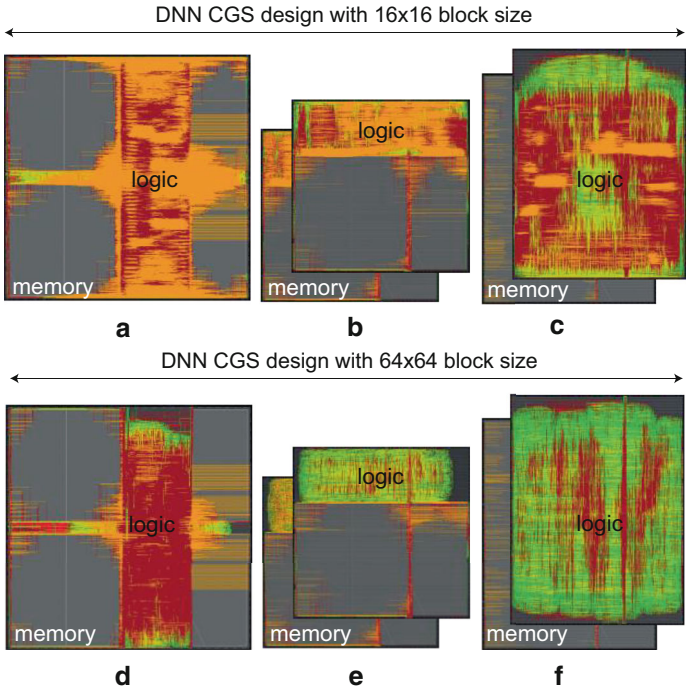


Fig. 34 GDS layouts of the implemented DNN CGS-16 and CGS-64 architectures at 400 MHz target clock frequency. DNN CGS-16 (**a**) 2D IC, (**b**) M3D-both, (**c**) M3D-one, DNN CGS-64, (**d**) 2D IC, (**e**) M3D-both, (**f**) M3D-one

Table 14 Iso-performance (400 MHz) design metric comparison of 2D and M3D ICs of DNN CGS-16 and CGS-64 architectures. All percentage values show the reduction from their 2D counterparts

Parameter	2D	M3D-both	Δ%	M3D-one	Δ%
DNN CGS-16					
Footprint (μm)	1411 × 1411	1010 × 984	−50.1%	996 × 1322	−33.9%
Wire-length (m)	12.089	8.469	−29.9%	12.225	1.1%
Cell count	298,309	262,084	−12.1%	290,692	−2.6%
Cell area (mm^2)	0.505	0.431	−14.6%	0.511	1.1%
Mem area (mm^2)	1.287	1.287	0.0%	1.287	0.0%
MIV count	–	77,536		1776	
Pin cap (pF)	943.3	788.0	−16.5%	1004.1	6.4%
Wire cap (pF)	2216.8	1440.8	−35.0%	2087.4	−5.8%
Total cap (pF)	3160.1	2228.7	−29.5%	3091.6	−2.2%
DNN CGS-64					
Footprint (μm)	1411 × 1411	1010 × 984	−50.1%	996 × 1322	−33.9%
Wire-length (m)	5.631	3.734	−33.7%	7.134	26.7%
Cell count	163,361	149,921	−8.2%	174,292	6.7%
Cell area (mm^2)	0.314	0.269	−14.3%	0.328	4.7%
Mem area (mm^2)	1.287	1.287	0.0%	1.287	0.0%
MIV count	–	48,636		1776	
Pin cap (pF)	520.8	390.8	−25.0%	553.5	6.3%
Wire cap (pF)	920.1	573.7	−37.7%	1110.5	20.7%
Total cap (pF)	1440.9	964.4	−33.1%	1664.0	15.5%

compared with the 2D ICs, whereas the M3D-one designs obtain only 33.9% reduction. This difference is attributed to the large memory area compared with logic in the DNN CGS-16 2D IC. These large memory blocks, if placed in the same tier, cause the footprint to increase significantly.

The wire-length saving reaches 29.9% and 33.7% in CGS-16 and CGS-64, respectively, with the M3D-both designs. This significant wire-length saving comes from the 50% smaller footprint and shorter distance among cells in M3D ICs. The M3D-both design for CGS-16 architecture achieves 12.1% cell count reduction, which leads to 14.6% total cell area saving. This saving mainly comes from fewer buffers and smaller gates needed to close timing in M3D ICs compared with the 2D counterparts. The savings in CGS-64 architecture are 8.2% and 14.3% for the cell count and area, respectively.

The 77 K MIVs are utilized in the CGS-16 architecture, while 48 K MIVs are used in CGS-64. This is mainly because CGS-16 design is more complex than CGS-64 (to be further discussed in later), so that the tier partitioning cutline cuts through more inter-tier connections in CGS-16. In the M3D-one design, logic and memory are separated into different tiers. This logic-memory connectivity is not high in the DNN architecture (= 1.7 K).

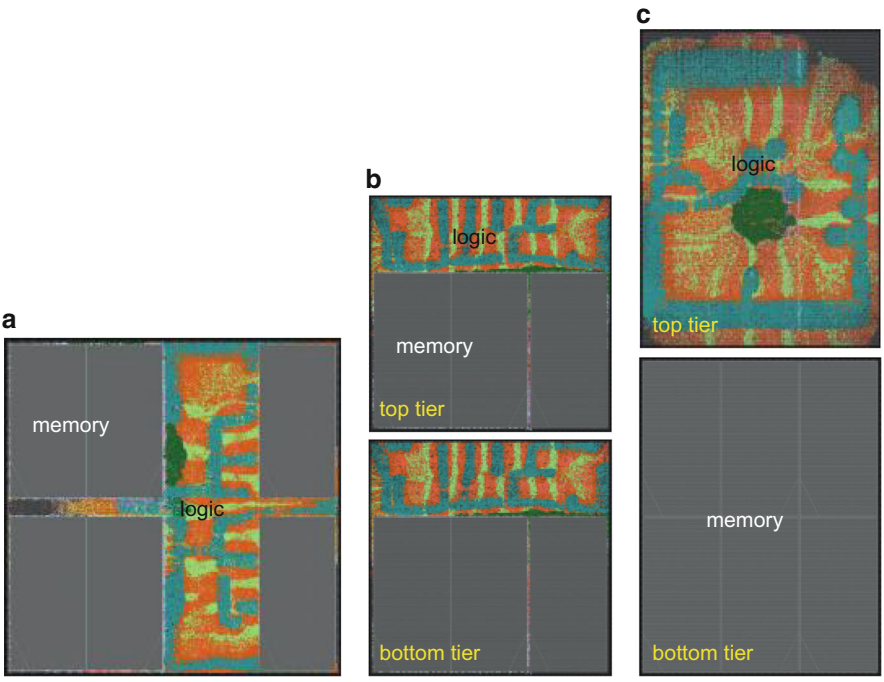


Fig. 35 Cell placement of the modules in CGS-16 architecture. (**a**) 2D, (**b**) M3D-both, (**c**) M3D-one. Each module is highlighted with different colors

In the CGS-16 architecture, the 16.5% pin capacitance saving is from cell area reduction, while the 35.0% wire capacitance saving is from wire-length reduction. By comparing the raw data, the DNN architecture is wire-dominated. The pin and wire capacitance saving reaches 25.0% and 37.7% in CGS-64.

To better understand why M3D-one gives significantly worse results than M3D-both, a placement comparison among 2D, M3D-both, and M3D-one designs is shown in Fig. 35. In the M3D-both design shown in Fig. 35b, the logic cells related to memory blocks in the top tier are placed in the same tier as the memory and densely packed to reduce wire-length effectively. This is the same for the bottom tier in the M3D-both design. On the other hand, logic gates are rather spread out across the top tier in the M3D-one design shown in Fig. 35c. This results in 1.1% increase in wire-length for CGS-16 and 26.7% increase in wire-length for CGS-64 compared with the 2D counterparts. This highlights the importance of footprint management and tier partitioning in the presence of large memory modules in DNN architectures.

Power Comparisons

Table 15 presents the iso-performance power comparison between 2D and M3D ICs of CGS-based DNNs. Internal, net switching, and leakage power breakdown is

Table 15 Iso-performance (400 MHz) power metric comparison of two architectures (CGS-16 vs. CGS-64) using two workloads (classification vs. pseudo-training). All percentage values show the reduction from their 2D counterparts

Workload	Power breakdown	2D	M3D-both	Δ%	M3D-one	Δ%
DNN CGS-16						
Classification	**Internal power (mW)**	91.3	76.7	−16.0%	90.3	−1.1%
	Net switching power (mW)	48.6	31.6	−35.0%	46.5	−4.3%
	Leakage power (mW)	1.3	1.2	−6.6%	1.3	0.5%
	Total power (mW)	141.1	109.6	−22.3%	138.0	−2.2%
Pseudo-training	**Internal power (mW)**	150.4	142.8	−5.1%	148.3	−1.4%
	Net switching power (mW)	68.4	57.1	−16.6%	65.6	−4.2%
	Leakage power (mW)	1.3	1.2	−6.8%	1.3	0.7%
	Total power (mW)	220.0	201.0	−8.6%	215.0	−2.3%
DNN CGS-64						
Classification	**Internal power (mW)**	86.8	76.1	−12.3%	84.9	−2.2%
	Net switching power (mW)	41.2	30.2	−26.7%	42.8	3.9%
	Leakage power (mW)	1.1	1.1	−4.7%	1.1	1.5%
	Total power (mW)	129.1	107.3	−16.9%	128.8	−0.2%
Pseudo-training	**Internal power (mW)**	129.2	120.0	−7.2%	128.5	−0.5%
	Net switching power (mW)	46.0	36.3	−21.2%	50.3	9.3%
	Leakage power (mW)	1.1	1.1	−4.6%	1.1	1.4%
	Total power (mW)	176.3	157.4	−10.7%	179.9	2.0%

reported for each design. The sign-off power calculations are conducted using two speech recognition workloads: classification and pseudo-training.

During classification, CGS-16 consumes 141.1 mW, while CGS-64 consumes 129.1 mW. This confirms that CGS-16 consumes more power to handle more complex weight selection process. A similar trend is observed during pseudo-training.

Pseudo-training, as expected, causes more switching in the circuits and thus more power consumption compared with classification for both CGS-16 and CGS-64 architectures.

Next, the power consumption of 2D and M3D ICs is compared. The resulting footprint of M3D-both designs is reduced by half, thereby reducing the wire-length between the cells. Figure 36a shows the wire-length distribution of the 2D and M3D ICs of CGS16 architecture. The histogram clearly shows that M3D ICs contain more number of short wires and fewer long wires compared with 2D IC. The effect of wire-length saving translates to the reduction of wire capacitance C_{wire} in Eq. 1, therefore the saving of the third term of the equation.

Figure 36b presents the distribution of standard cells with different ranges of cell drive-strength. M3D-both design uses more number of low drive-strength cells (i.e., $\times 0 \sim \times 0.8$) and fewer high drive-strength cells (i.e., $\times 1 \sim \times 16$). Since low drive-strength cells utilize smaller transistors, the short circuit current of the transistors and C_{pin} are lower, which reduces both the first and second term in Eq. 1.

Impact of Monolithic 3D ICs on Performance of Deep Neural Network Hardware

In this section, the impact of M3D stacking technology on the performance of CGS-16 and CGS-64 architectures is investigated by pushing the target clock frequency of 2D and M3D ICs to their maximum clock frequency. 2D and M3D ICs are implemented with TSMC® 28 nm HPM technology sweeping the target frequency from 400 MHz in 25 MHz increments. The floorplans of the 2D and M3D ICs are same as the ones used in the previous section.

As the M3D-both designs show better design quality compared to the M3D-one designs, memory blocks are placed on both tiers in the M3D ICs for this experiment as shown in Fig. 37.

The maximum performance comparison between the 2D and M3D ICs of CGS-16 and CGS-64 architectures is presented in Table 16. The table shows the target clock frequency used to place-and-route the designs, the resulting WNS from static

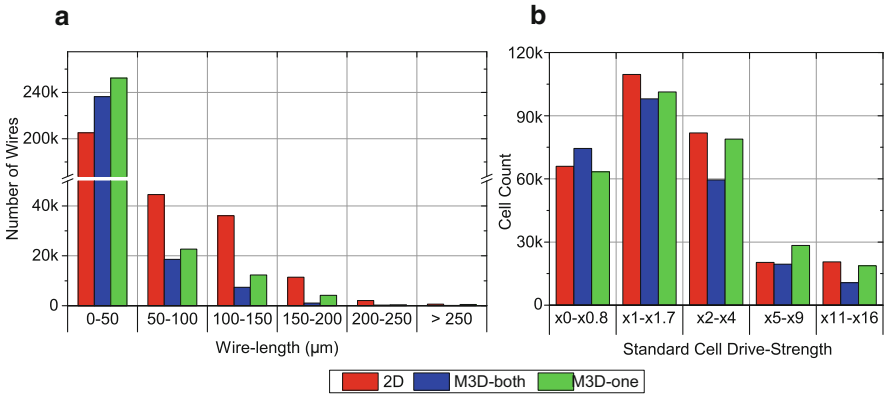


Fig. 36 (**a**) Wire-length and (**b**) cell drive-strength distribution of DNN CGS-16 2D, M3D-both, and M3D-one

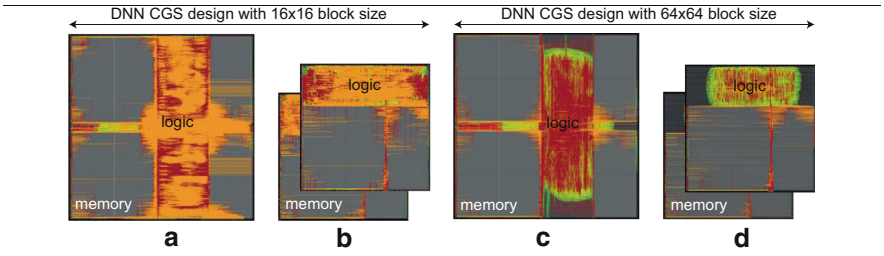


Fig. 37 GDS layouts of 2D and M3D ICs of DNN CGS-16 and CGS-64 architectures at the maximum target frequencies. (**a**) 2D IC at 550 MHz, (**b**) M3D IC at 575 MHz of DNN CGS-16 architecture, (**c**) 2D IC at 600 MHz, (**d**) M3D IC at 625 MHz of DNN CGS-64 architecture

timing analysis, and the effective clock frequency, which is the maximum achievable clock frequency that the designs are able to operate at without timing violation.

Comparing only the 2D ICs of CGS-16 and CGS-64 architectures, the effective clock frequency of the CGS-16 2D IC is 11.1% less than the CGS-64 2D IC. As the critical path of the CGS-16 2D IC starts from weight SRAM to MAC unit through weight selection logic, the lower effective clock frequency of the CGS-16 2D IC is attributed to its more complex weight selection logic as shown in a higher design density in Fig. 37a compared to Fig. 37c.

Next, the maximum performance of the 2D and M3D ICs is compared. The M3D ICs shows 6.2% and 1.2% performance improvement over 2D counterparts in CGS-16 and CGS-64 architectures, respectively. To analyze this trend, the worst timing path comparison of the 2D and M3D ICs is conducted. Fig. 38 compares the same timing path (i.e., the worst timing path of the 2D IC) in the 2D and M3D CGS-16 designs at the maximum target clock frequency of the 2D IC, and Table 17 presents key metrics of the timing path.

The wire-length of the worst timing path of the 2D IC is 53.6% longer than the same timing path in the M3D IC. This is attributed to the reduced footprint and the

Table 16 Maximum performance comparison of 2D and M3D ICs of DNN CGS-16 and CGS-64 architectures

Parameter		DNN CGS-16	DNN CGS-64
2D	**Target clk freq (MHz)**	550	600
	WNS (ns)	−0.056	0.002
	Effective clk freq (MHz)	534	601
M3D	**Target clk freq (MHz)**	575	625
	WNS (ns)	−0.024	−0.046
	Effective clk freq (MHz)	567	608
Δ% effective clk freq		6.2%	1.2%

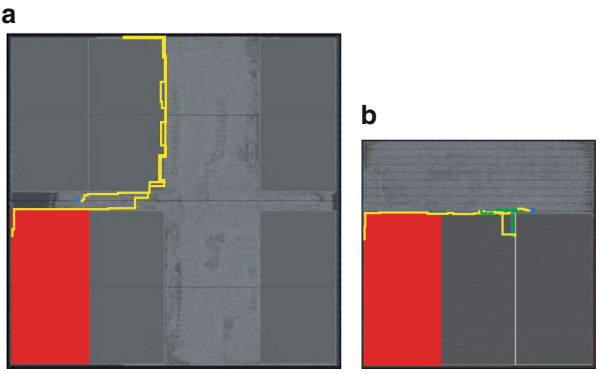


Fig. 38 Worst timing path comparison of 2D and M3D ICs of DNN CGS-16 architecture. (**a**) The worst timing path of 2D IC at its maximum target clock frequency, 550 MHz. (**b**) The same timing path in M3D IC

Table 17 Key parameter comparison of the worst timing path in Fig. 38 of the 2D and M3D ICs of DNN CGS-16 architecture

Parameter	2D	M3D	Δ%
Wire-length (μm)	3208	1488	−53.6%
Cell count	65	49	−24.6%
Avg. cell drv-str	8.9	7.0	−21.3%
Cell area	180.1	66.4	−63.1%
MIV count	–	6	
Wire cap (fF)	500	242	−51.6%
Pin cap (fF)	486	312	−35.8%
Resistance (kΩ)	14.5	9.3	−35.9%
Delay (ns)	2.344	2.088	−10.9%

inter-tier connections of the M3D IC, which results in shorter distance among cells along the timing path. The M3D IC offers 24.6% cell count saving as well as 21.3% average cell drive-strength reduction, thereby reducing cell area by 63.1% of the timing path. This is because the fewer and smaller buffers are needed to drive the reduced wire-load, which is a result of the wire-length reduction.

Compared to the 2D IC, the wire and pin capacitance of the timing path in the M3D IC are reduced by 51.6% and 35.8%, respectively. The wire capacitance reduction mainly comes from the wire-length reduction of the timing path, whereas the pin capacitance saving results from the cell count and cell drive-strength reduction. In addition, the M3D IC achieves 35.9% resistance reduction in the timing path. The resistance saving is also attributed to the wire-length saving along the timing path.

Due to the capacitance and resistance saving of the worst timing path, the delay of the timing path is reduced by 10.9% in the M3D IC, thereby offering rooms to improve the performance.

In order to understand the impact of the above observations to the overall timing paths of the 2D and M3D ICs, the slack distribution of all timing paths of the 2D and M3D CGS16 designs is reported in Fig. 39. While 18 timing paths of the 2D IC violate the timing constraints, the M3D IC successfully closes timing without any violation. In addition, there are more timing paths with high positive slack in the M3D IC, which indicates that timing is easily closed in the M3D IC due to the reduced delay of the timing paths.

The difference in the performance improvement of the M3D ICs of CGS-16 and CGS64 architecture is be discussed in detail in the next section.

Architectural Impact Discussions

CGS-16 and CGS-64 Architecture Comparisons

Table 15 shows that the total power reduction of M3D ICs is higher in DNN CGS-16 architecture than CGS-64. Furthermore, more performance improvement with M3D ICs is achieved in DNN CGS-16 architecture as shown in Table 16. These differences are caused by the granularity of weight selection methodology (i.e., CGS

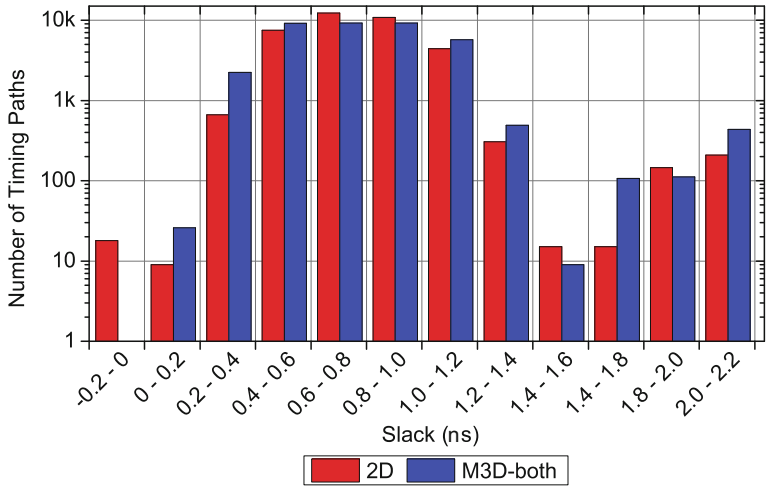


Fig. 39 Slack distribution comparison between 2D and M3D ICs of DNN CGS-16 architecture at the maximum clock frequency of the M3D IC

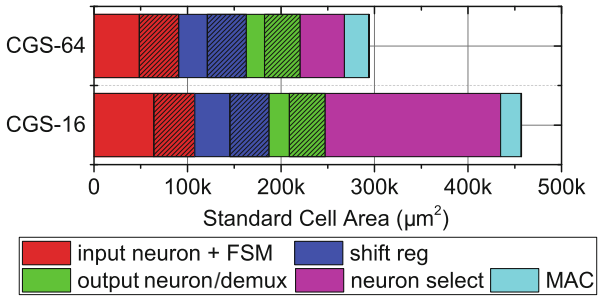


Fig. 40 Standard cell area breakdown of 2D CGS-16 and CGS-64 architectures. Non-dashed and dashed boxes, respectively, indicate combinational and sequential elements. Only five largest modules are shown

algorithm). The 1024 × 1024 weight matrix is divided into 256 (= 16 × 16) weight blocks in CGS-64 architecture. This count becomes 4096 (= 64 × 64) weight blocks in CGS-16. The implication in DNN architecture is that CGS-16 requires a more complex neuron selection unit than CGS-64. Figure 40 shows the comparison of standard cell area of each module in CGS-16 and CGS-64 architectures. Both sequential (dashed box) and combinational logic (non-dashed box) portion in each module are shown. The neuron selection unit in CGS-16 architecture (shown in purple) occupies more area than that in CGS-64 architecture.

As discussed before, M3D ICs benefit not only from wire-length reduction but also from standard cell area saving. The number of storage elements (i.e., sequential logic and memory blocks) used in 2D and M3D ICs remains the same. Thus, the only possible power reduction coming from storage elements is their

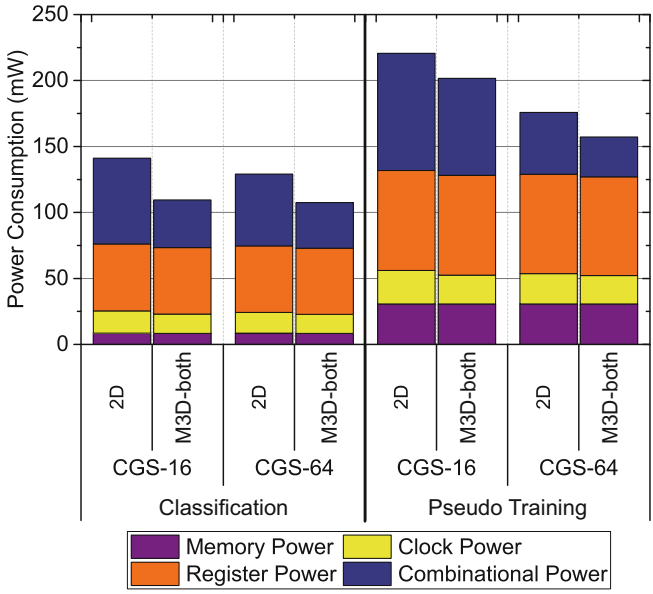


Fig. 41 Power breakdown under two DNN architectures (CGS-16 and CGS-64), two workloads (classification and pseudo-training), and two designs (2D and M3D ICs)

drive-strength reduction. This does not show a huge impact considering the small portion of sequential elements in the DNN architectures (16.1% on average). On the other hand, combinational logic can be optimized in various ways, such as logic reconstructing and buffer reduction. Therefore, the DNN M3D ICs benefit more from combinational logic gates than sequential elements.

Figure 41 shows the breakdown of total power consumption into combinational, register, clock, and memory portions. Combinational power reduction is the dominant factor in total power saving of M3D ICs in both CGS-16 and CGS-64 architectures and in both classification and pseudo-training workloads. The saving in other parts including register, clock, and memory power largely remains small. In addition, the neuron selection unit in CGS-16 architecture consists of a larger number of combinational logic gates than CGS-64. Thus, its M3D ICs have more room for power optimization, resulting in a larger combinational power saving.

The larger neuron selection logic in CGS-16 architecture also offers more opportunity to improve the performance of M3D ICs. While 2D ICs suffer long timing path due to the complex neuron selection logic, M3D ICs effectively reduce the wire-length, providing buffer count/size reduction along the worst timing path. This reduces the capacitance and resistance of timing paths, thereby offering shorter delay and larger performance improvement.

Figure 42 compares the total wire-length and standard cell count along the selected 486 timing paths, which are from weight SRAMs to registers of MAC units through neuron selection logic, in the 2D/M3D CGS-16/CGS-64 designs at

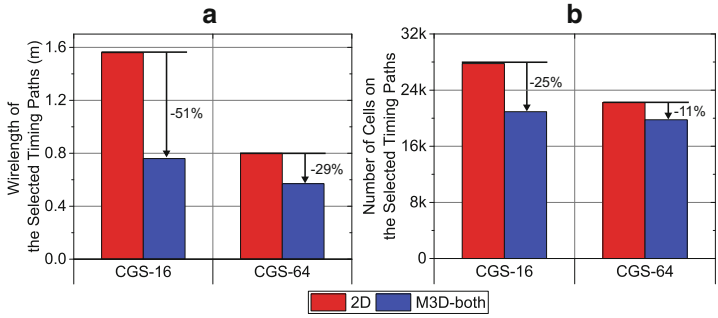


Fig. 42 Comparison of (**a**) the wire-length and (**b**) cell count of the timing paths from weight SRAMs to registers in MAC units through neuron selection logic in 2D and M3D both of DNN CGS-16 and CGS-64 architecture

the maximum frequency of the 2D ICs. Comparing only the 2D ICs, the CGS-16 2D IC clearly utilizes longer wire-length as well as more standard cells as the neuron selection logic is more complex. As the CGS-16 M3D IC has more combinational logics to optimize with the reduced footprint, it offers more cell count and wire-length reduction compared to the CGS-64 M3D IC, providing more rooms for performance improvement in higher clock frequency.

Impact of Workloads

In order to investigate the impact of different DNN workloads on M3D power saving, two main types of speech recognition DNN workloads are analyzed: feed-forward classification and training. Real-world test vectors are used for feed-forward classification. However, since the current architecture does not support online training to avoid computational overhead of finding gradients in DNN training, customized test vectors are created for "pseudo-training." Online training on DNN consists of feed-forward computation and backward computation. In order to mimic the online training on the current architecture, there are two phases in the pseudo-training test vectors as shown in Fig. 43. In the first phase, the DNN performs feed-forward classification, which represents feed-forward computation during training. In the second phase, the DNN conducts feed-forward classification and writes the weights to memory blocks, which represents backward computation and weight update. These two phases mimic the behavior of logic computation and weight update during training.

Table 15 shows that while M3D-both shows 22.3% (CGS-16) and 16.9% (CGS-64) total power reduction in feed-forward classification workload, the power saving of pseudo-training workload is only 8.6% (CGS-16) and 10.7% (CGS-64). This difference stems from different switching patterns of combinational logic and storage elements in the DNN architecture. The DNN mainly uses combinational logic gates to compute the values of neuron outputs and access memory for read operations only during feed-forward classification. Thus, this workload is classified as a compute-intensive kernel. On the other hand, memory operations are heavily

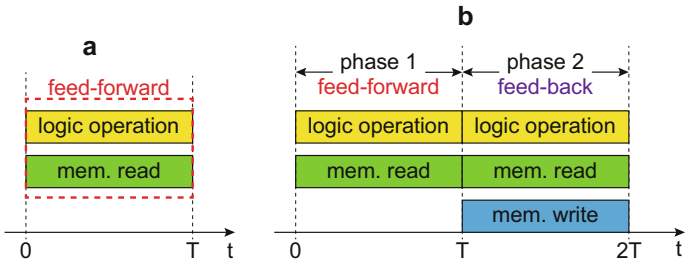


Fig. 43 Comparison of the operations in (**a**) the feed-forward classification and (**b**) pseudo-training

used during pseudo-training since the DNN architecture needs to read and write weights. This becomes a memory-intensive kernel. Therefore, switching activity in memory blocks is much higher during pseudo-training while that of combinational logic remains largely similar. This explains larger power consumption during pseudo-training workload: 220.0 mW vs. 141.1 mW for CGS-16 and 176.3 mW vs. 129.1 mW for CGS-64 as shown in Table 15.

As shown in Fig. 41, memory power and register power occupy a large portion of the total power during pseudo-training. This means that the combinational logic power saving becomes a smaller portion of the total power saving during training. The opposite is true for classification, where memory and register power are less dominant. In this case, the reduction in combinational power saving becomes more prominent in the total power saving.

Conclusion

As device scaling in advanced technology nodes is slowly saturating due to low volume and yields, M3D stacking technology has come into the spotlight as an alternative for continuing Moore's low, showing its strength in reducing power consumption and enhancing performance utilizing short vertical connections among tiers instead of using long wires on the xy-plane. However, there are numerous challenges on M3D stacking technology mainly because of its technological prematurity, sequential fabrication process, and high vertical integration density.

In this chapter, in-depth analysis on the challenges is performed, and CAD tool solutions are presented.

First, a comprehensive study is performed investigating the power impact of M3D ICs using a commercial in-order 32-bit application processor implemented on foundry 28 nm, foundry 14/16 nm and 7 nm technology nodes. M3D ICs provide maximum power savings at the 28 nm technology node. The benefits improve at higher clock frequencies with the reduction of standard cell area in addition to wire-length savings. Based on the findings, a M3D IC design flow, called cascade-2D design flow, is presented to implement M3D ICs using 2D commercial

tools. Cascade-2D design flow utilizes a design-aware partitioning scheme where functional modules with very large number of connections are partitioned into separate tiers. The MIVs are modeled as sets of anchor cells and dummy wires, which enable to implement and optimize both top and bottom tiers simultaneously in a 2D IC. Cascade-2D design flow reduces standard cell area effectively, resulting in significantly better power savings than existing M3D IC design flows. Additionally, by leveraging smaller standard cells, M3D ICs can save die area which directly translates to reduced costs.

Next, an in-depth study on PDNs in M3D ICs is presented. A system-level PDN of M3D ICs is built, and comprehensive studies including static/dynamic rail analysis as well as frequency-/time-domain analysis are performed. Although M3D PDNs suffer from high IR-drop due to additional metal layers, irregular placement of power MIVs, and fewer C4 bumps, they reduce $L\mathrm{d}i/\mathrm{d}t$-drop from 3D placement of decap cells. Additionally, higher resistance of M3D PDN due to its series resistive path across tiers improves the resiliency against AC noise showing peak impedance reduction at first-order resonance frequency.

For DNN M3D ICs, the impact of M3D stacking technology on power, performance, and area is examined with speech recognition DNN architectures that exhibit coarse-grain sparsity. M3D ICs reduce the total power consumption more effectively with compute-intensive workloads, compared to memory-intensive workloads. By placing memory blocks evenly on both tiers, M3D ICs reduce the total power consumption significantly. In addition, owing to the reduced footprint and vertical integration, M3D ICs offer performance improvement over 2D ICs, especially in architecture with complex combinational logics. This convincingly demonstrates the low-power and high-performance benefits of M3D ICs on DNN hardware and offers architectural guidelines to maximize the benefits.

References

Arden W, Brillouët M, Cogez P et al (2012). More-than-Moore, A White Paper. In IEEE international roadmap for devices and systems, p 31

Batude P, Fenouillet-Beranger C, Pasini L et al (2015) 3DVLSI with CoolCube process: an alternative path to scaling. In Proc. symp. on VLSI technology, 2015

Billoint O, Sarhan H, Rayane I et al (2015) A comprehensive study of monolithic 3D cell on cell design using commercial 2D tool. In: Proc. design, automation and test in Europe, 2015

Bozorgzadeh B, Afzali-Kusha A (2008) Decoupling capacitor optimization for nanotechnology designs. In: Proc. int. conf. on microelectronics, 2008

Chang K, Acharya K, Sinha S et al Impact and design guideline of monolithic 3-D IC at the 7-nm technology node IEEE Trans VLSI System, vol. 25, p. 2118–2129, July 2017

ChengY, Yu FX, Feris RS et al (2015) An exploration of parameter redundancy in deep networks with circulant projections. arXiv:1502.03436 [cs], February 2015

Cheng Y, Wang D, Zhou P (2017) A survey of model compression and acceleration for deep neural networks. arXiv:1710.09282 [cs], October 2017

Conneau A, Kiela D, Schwenk H et al (2017) Supervised learning of universal sentence representations from natural language inference data. arXiv:1705.02364 [cs], May 2017

Courbariaux M, Bengio Y, David J-P (2015) Binary connect: training deep neural networks with binary weights during propagations. arXiv:1511.00363 [cs], November 2015

Courbariaux M, Hubara I, Soudry D (2016) Binarized neural networks: training deep neural networks with weights and activations constrained to +1 or −1. arXiv:1602.02830 [cs], February 2016

Das S, Whatmough P, Bull D (2015) Modeling and characterization of the system-level power delivery network for a dual-core aRM cortex-A57 cluster in 28 nm CMOS. In: Proc. int. symp. on low power electronics and design, 2015

Deng L, Hinton G, Kingsbury B (2013) New types of deep neural network learning for speech recognition and related applications: an overview. In: Proc. int. conf. on acoustics, speech and signal processing, 2013

Gardner WA (1984) Learning characteristics of stochastic-gradient-descent algorithms: a general study, analysis, and critique. Signal Process 6:113–133

Garofolo JS, Lamel LF, Fisher WM et al (1993) DARPA TIMIT acoustic-phonetic continous speech corpus CD-ROM. NIST Speech Disc 1-1.1, NASA STI/Recon technical report N, vol. 93, February 1993

Graves A, Mohamed A-r, Hinton G (2013) Speech recognition with deep recurrent neural networks. arXiv:1303.5778 [cs], March 2013

Gray S, Radford A, Kingma DP (2017) GPU kernels for block-sparse weights

Han S, Mao H, Dally WJ (2015) Deep compression: compressing deep neural networks with pruning, trained quantization and huffman coding. arXiv:1510.00149 [cs], October 2015

Han S, Kang J, Mao H et al (2016) ESE: efficient speech recognition engine with sparse LSTM on FPGA. arXiv:1612.00694 [cs], December 2016

He T, Fan Y, Qian Y et al (2014) Reshaping deep neural network for fast decoding by node-pruning. In: Proc. int. conf. on acoustics, speech and signal processing, 2014

He K, Zhang X, Ren S et al (2015) Deep residual learning for image recognition. arXiv:1512.03385 [cs], December 2015

Kadetotad D, Arunachalam S, Chakrabarti C et al (2016) Efficient memory compression in deep neural networks using coarse-grain sparsification for speech applications. In: Proc. int. conf. on computer-aided design

Karpathy A, Fei-Fei L (2017) Deep visual-semantic alignments for generating image descriptions. IEEE Trans Pattern Anal Machine Intell 39:664–676

Khan NH, Alam SM, Hassoun S (2011) Power delivery design for 3-D ICs using different through-silicon via (TSV) technologies. IEEE Trans VLSI System 19:647–658

Krizhevsky A, Sutskever I, Hinton GE (2012) ImageNet classification with deep convolutional neural networks. In: Proc. int. conf. on neural information processing systems, 2012

Larsson P (1998) Resonance and damping in CMOS circuits with on-chip decoupling capacitance. IEEE Trans Circuits System 45:849–858

Liao W, He L, Lepak KM (2005) Temperature and supply voltage aware performance and power modeling at microarchitecture level. IEEE Trans Comput-Aided Design Integr Circuits Syst 24:1042–1053

Liao S, Li Z, Lin X et al (2017) Energy-efficient, high-performance, highly-compressed deep neural network design using block-circulant matrices. In: Proceedings of IEEE international conference on computer aided design

Okada M, Sugaya I, Mitsuishi H et al (2014) High-precision wafer-level Cu-Cu bonding for 3DICs. In: Proc. int. electron devices meeting, 2014

Pant S, Chiprout E (2006) Power grid physics and implications for CAD. In: Proc. design automation conf. 2006

Panth SA, Samadi K, Du Y, Lim SK (2014) Design and CAD methodologies for low power gate-level monolithic 3D ICs. In: IEEE international symposium on low power electronics and design, 2014

Povey D, Ghoshal A, Boulianne G et al (2011) The kaldi speech recognition toolkit. IEEE workshop on automatic speech recognition and understanding, January 2011

Seo KI, Haran B, Gupta D et al (2014) A 10nm platform technology for low power and high performance application featuring FINFET devices with multi workfunction gate stack on bulk and SOI. In: Proc. symp. on VLSI technology, 2014

Song T, Rim W, Jung J et al (2015) A 14 nm FinFET 128 Mb SRAM with Vrm MIN enhancement techniques for low-power applications. IEEE J Solid State Circuits 50:158–169

Su D, Wu X Xu L (2010) GMM-HMM acoustic model training by a two level procedure with gaussian components determined by automatic model selection. In: Proc. int. conf. on acoustics, speech and signal processing

Sze V, Chen Y-H, Emer J et al (2017) Hardware for machine learning: challenges and opportunities. arXiv:1612.07625 [cs], April 2017

Wu SY, Lin CY, Chiang MC et al (2013) A 16 nm FinFET CMOS technology for mobile SoC and computing applications. In: Proc. int. electron devices meeting, 2013

Xiong W, Droppo J, Huang X et al (2016) The Microsoft 2016 conversational speech recognition system. arXiv:1609.03528 [cs], September 2016

Yang SH, Sheu JY, Ieong MK et al (2011) 28nm metal-gate high-K CMOS SoC technology for high-performance mobile applications. In: Proc. custom integrated circuits conf., 2011

Part V
Processor Design and Programming Flows

Architecture Description Languages

23

Anupam Chattopadhyay, Zheng Wang, and Grant Edmund Martin

Contents

Introduction . . . 808
A Brief History of ADLs . . . 812
The Classical Era: 1990–2000 . . . 813
The First Industrial Era: 2000–2010 . . . 813
The Second Industrial Era: 2010–2020 . . . 814
Types and Characteristics of ADLs . . . 814
Types of ADLs . . . 814
Characteristics of ADLs . . . 815
Key ADLs . . . 816
MIMOLA . . . 816
EXPRESSION . . . 817
nML . . . 817
LISA . . . 818
PEAS . . . 819
TENSILICA TIE . . . 819
ARC APEX . . . 820
Codasip CodAL . . . 821
Andes ACE . . . 822
RISC-V Chisel . . . 822

A. Chattopadhyay
School of Computer Science and Engineering, Nanyang Technological University, Singapore, Singapore
e-mail: anupam@ntu.edu.sg

Z. Wang (✉)
Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, Shenzhen, China
e-mail: zheng.wang@siat.ac.cn

G. E. Martin
Independent Consultant, Pleasanton, CA, USA
e-mail: gmartin@ieee.org

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_18

ADL-Driven Methodologies 823
Generation of Software Tools 823
Automatic Synthesis of Custom Instructions for an Application 824
Instruction-Set Simulator Generation 825
Generation of Hardware Implementation 827
Top-Down Verification 828
Validation of an ADL Specification 829
Specification-Driven, Simulation-Based, Verification 830
Applications of ADL-Based Design 831
Conclusions 833
References 834

Abstract

Designing a processor is an arduous task. It involves not only defining the instruction-set architecture but also the various processor development tools such as compiler, instruction-set simulator, debugger, assembler, and linker. Moreover, an efficient hardware implementation of the entire microarchitecture is needed apart from verifying the entire design at every step while meeting various end user constraints. These tasks were traditionally done by various specialized teams, originating from an informal processor specification, which posed a significant challenge of maintaining design performance and consistency. The emergence of Architecture Description Languages (ADLs) addressed this challenge. In this chapter, the background of processor design automation flows and how ADLs fit into this narrative are discussed. Prominent ADLs, including commercial ones, are summarized to provide an idea about the inner workings of ADL-driven design flows.

Keywords

Processor architecture · Embedded systems · Application-specific instruction-set processors · ASIPs · Architecture description languages · ADLs · Instruction-set architectures · ISAs · Processor description languages

Introduction

The development of electronics since the middle of the last century has been notable for many advances in technology, but one in particular was described by Tsugio Makimoto of Sony in Japan in 1991: Makimoto's Wave (Makimoto 2002). In this paper, Makimoto describes how the semiconductor industry has swung on a cyclical basis between standardization and customization, in cycles lasting about a decade. When one considers computing devices in the most general terms, from the mainframe computer through to the modern era of both desktops, laptops, and servers as one pole, and embedded computing devices found ubiquitously around everyone in appliances, phones, wearable devices, vehicles, and the "Internet of Things (IoT)," one can see very similar waves.

The earliest mainframe era, for example, featured many different types of processors with their associated Instruction Set Architectures (ISAs) – so, a lot of customization. But the IBM 360 ISA became dominant in the marketplace – so was standardized in many uses. Minicomputers followed the same trend, with a variety of vendors, but the DEC ISAs becoming dominant. The advent of the microprocessor led to a wide variety of processor architectures and ISAs, (customization), but two in particular became dominant: Intel's x86 ISA, for desktops, laptops, and servers, and ARM's embedded processors for phones and many other devices. Hence, customization was followed by standardization.

The design and verification of new processor ISAs in the earliest days required large engineering teams. These included large hardware teams and also specialized software teams to create the tools required to program and debug these processors. As ISAs swung from Complex Instruction Set Computers (CISCs) through to generations of Reduced Instruction Set Computers (RISCs), the role of the optimizing compiler grew ever more important so that programmers could develop the increasing amount of application software required to allow the proliferation of computing into all aspects of life (see ▶ Chap. 32, "Retargetable Compilation"). Proliferation of programming languages at various levels of abstraction made the software tools crisis ever more in need of a solution. The tools available in both hardware and software domains in the early periods were primitive and led in themselves to considerable research and development of electronic design automation (EDA) tools for hardware design and verification and new programming models, abstractions, and tools in the software domain.

All these computing devices use programmable components: processors, coprocessors, and relatively less-programmable components, such as in-memory computing accelerators and Tensor Processing Units (TPUs). These programmable components are also generally referred as *programmable accelerators*. Figure 1 shows a typical embedded system with various programmable accelerators. In any chosen application domain, the embedded system can have application-specific programmable accelerators, reconfigurable logic fabrics, general-purpose graphics processing units, digital signal processors, image/audio/cryptographic co-processors, specialized processors for Artificial Intelligence/Machine Learning ("AI/ML") computations, communication buses, and peripherals. The variety and complexity of these programmable accelerators are increasing at a rapid pace due to the growing demand for design of increasingly complex applications in smart healthcare, smart infrastructure, ubiquitous communication, and in general applications driven by growth of communication, sensing, and intelligence capabilities. This push for complex programmable accelerator design is further made challenging due to shrinking time-to-market as well as short product lifetimes. This calls for an efficient and effective design automation flow for complex programmable accelerators (often known as Application-Specific Instruction-set Processors, or ASIPs).

The most crucial roles in the design automation of ASIPs are played by **specification** and **modeling**. It is imperative to develop a high-level and sufficiently expressive specification language that can model the complex processors and their ISAs. The specification language needs to enable automated design performance

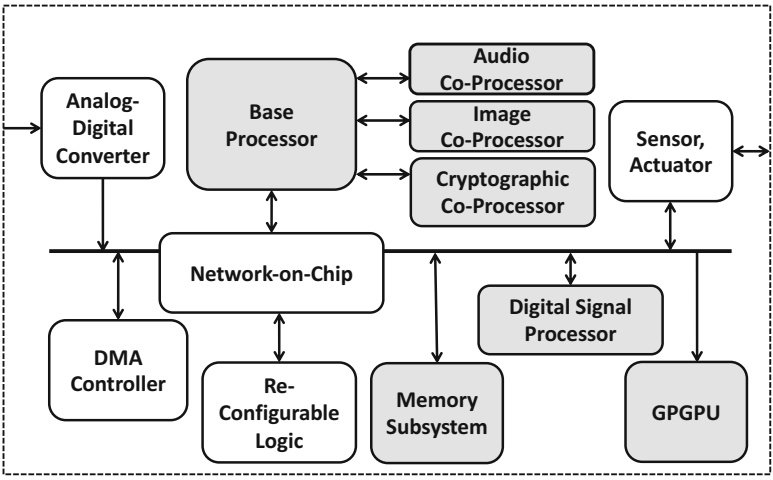


Fig. 1 Example embedded system showing a variety of components including programmable accelerators

analysis and eventual generation of efficient final implementations and verification. The language needs to be expressive enough for capturing high-level descriptions of the programmable architectures. At the same time, the language needs to be simple enough to be usable by a large number of design teams ranging in skill from students through to application-specific product design teams with years of industrial experience. It then needs to produce the artifacts required for the application programming teams to efficiently exploit the resulting application-specific accelerators.

These observations led researchers to work towards Architecture Description Languages (ADLs) and ADL-driven design automation flows (Leupers et al. 2016; Gries and Keutzer 2005), some of which have made successful transitions to industry. The term "Processor Description Language" (Mishra and Dutt 2008) has also been used, although ADL is more common.

Using natural language specifications written in languages such as English for processor specification was only feasible when concentrated into large design teams led by experienced ISA architects or researchers, who could resolve ambiguities, incompleteness, and errors as they arose. Only in this way could they guarantee consistent interpretations of the same specification. Even in this case, turnover, geographic distribution of design teams, and the split between design and verification engineers, and hardware and software teams, often led to serious errors caught either by luck, huge amounts of verification runtime, or not caught at all until prototypes or products were built and, in the field, revealed the presence of serious errors and issues.

Though formal specification languages are much more suitable for analysis and verification, for a long time, there was no serious research effort towards formal

processor description languages. A few formal specification languages were used as the input languages for powerful verification tools such as a model checker. However, these specifications cannot be used by designers or other tool developers, for what was still, largely, a manual effort. An ideal specification language should have formal (unambiguous) semantics to support the verification flows as well as automated generation of artifacts required to use the processor. ADLs addressed this exact research and development problem and, since the earliest ADLs, have been successfully applied.

Development of a processor is associated with multiple steps: *benchmarking*, *architectural space identification*, *design evaluation*, *design exploration, design verification,* and *deployment*. The Mescal methodology (Gries and Keutzer 2005) is a good example of this kind of flow and the way in which an ADL-based specification can be used for design representation, design evaluation, design validation, and synthesis from a higher abstraction to a more detailed abstraction such as Register Transfer Level (RTL), and the generation of associated software tools. This is depicted in Fig. 2.

The ADL specification and tool flow is used to automatically create the associated processor software tools, such as the instruction-set simulator (ISS), compiler, profiler, debugger, assembler, and linker. The specification is also used by the tool flow to generate a detailed and optimized hardware description in RTL format (Mishra et al. 2004; Schliebusch et al. 2004a). The ADL specification is utilized

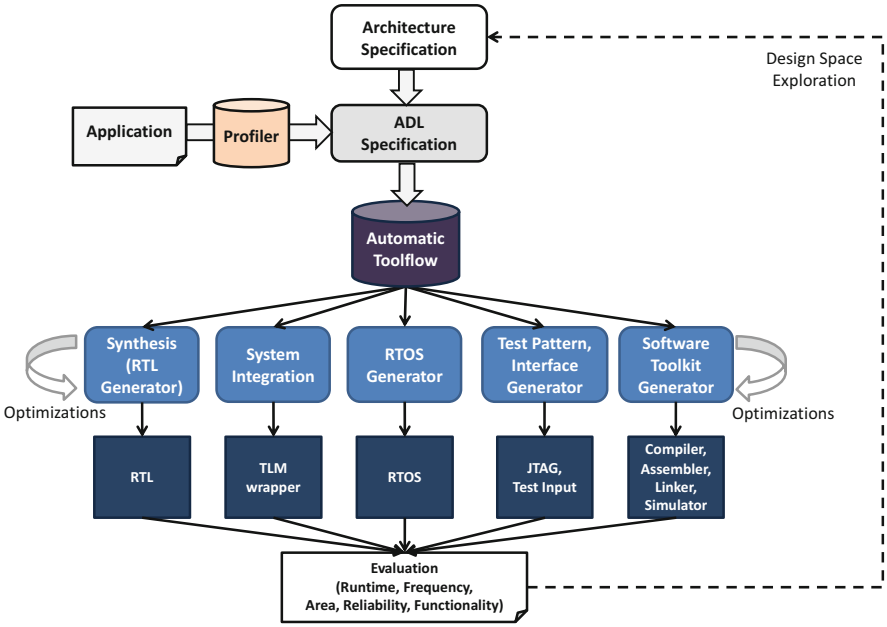


Fig. 2 ADL-driven exploration, synthesis, and validation of programmable architectures

to validate the design by formal, semi-formal, and simulation-based validation (Mishra and Dutt 2005a) as well as for the automated generation of test interfaces (Schliebusch et al. 2004b). Sometimes such a specification can be used to generate device drivers (Wang and Malik 2003), although is less common. Some ADL-based tool flows can instead generate hardware abstraction layer routines for a particular ISA that can be embedded in a skeletal RTOS and modified by users for their own uses (Augustine et al. 2009).

The specification and modeling capabilities of ADLs have been used to implement many different kinds of processor architectures ranging from programmable coarse-grained reconfigurable architectures, VLIW processors, to superscalar processors (Mishra et al. 2004; Rakossy et al. 2012, 2013). Early-stage design space exploration using an ADL-driven tool suite has been extended to cover high-level power and reliability estimations (Wang and Chattopadhyay 2017). Naturally, with the increasing variety and complexity of ASIPs, shortening time to market, and the growth of modeling efficiency in the ADLs, there has been a steady adoption of ADL-based design automation flows by the industry (Synopsys ASIP Designer; Cadence Tensilica). This chapter presents an overview of processor design tools, from the perspective of ADL-based design methodology.

The rest of the chapter is organized as follows. Section "A Brief History of ADLs" presents a brief history of ADLs. Section "Types and Characteristics of ADLs" covers the major types of ADLs and their key characteristics, thus illustrating how they are used for processor specification and modeling. Section "Key ADLs" briefly outlines several of the key ADLs that have emerged during the history of this technology. Then section "ADL-driven Methodologies" dives into detail about several ADL-driven methodologies. A variety of applications for which ADLs have been successfully used are reviewed in section "Applications of ADL-based Design". Finally, section "Conclusions" concludes the chapter with glimpses of ongoing and future research agendas.

A Brief History of ADLs

The history of ADLs can be divided into three eras, each approximately a decade or so in length. These are:

1. The Classical era (1990–2000), in which academic researchers developed the basic underlying concepts for ADLs and the initial sets of tools and flows to support processor design with ADLs.
2. The First Industrial era (2000–2010), in which some of the academic ADLs became the basis for commercial tool application, and some commercial ADLs emerged without having any basis in academic research. There were also industrial mergers of some of the earliest ADL companies with others.
3. The Second Industrial era (2010–2020), in which the tools from the first industrial era persisted in commercial use, albeit with more mergers and

changes. There were also significant arrivals of new ADLs from new commercial providers, triggered in part by the RISC-V movement. Except for RISC-V, which arose in academia, most of the academic interest in ADL research declined. The growth in RISC-V architectures is also timed with a range of serious vulnerabilities identified in commercial processors. Thus, RISC-V architectures and its associated design flows do consider security as a fundamental design objective (Watson et al. 2019).

In addition to the brief descriptions of ADLs in this chapter, the reader might look to several comprehensive ADL surveys available in the literature including ADLs for retargetable compilation (Qin and Malik 2002), programmable embedded systems (Mishra and Dutt 2005b), and SoC design (Tomiyama et al. 1999). A definitive compilation of the ADLs can be found in references (Leupers et al. 2016; Mishra and Dutt 2008).

The Classical Era: 1990–2000

In the classical era, there were notable academic research results developed in ADLs and tool flows. Motivations included a desire to raise the EDA levels of abstraction to higher levels to better automate processor design. Thus, there is a natural basis in the ADL work to evolve from RTL abstractions and to focus more on hardware generation. Examples of early research ADLs are Mimola, EXPRESSION, nML, LISA, and PEAS, which are described below.

The First Industrial Era: 2000–2010

This era saw two major developments: a transition from academic research to industrial application for some ADLs such as LISA and nML, and the development within industry of ADL-based ASIP design tools such as Tensilica TIE, and ARC APEX.

The industrial history based both on within-industry developments and academic transfers can be complex and hard to follow. nML, for example, was commercialized by Target Compiler Technologies in 1996, later purchased by Synopsys in 2014. LISA spun out of the Institute for Integrated Signal Processing Systems (ISS) of RWTH Aachen as a company, LISATek, in 2001. LISATek was then bought by CoWare in 2003; CoWare itself was acquired by Synopsys in 2010.

ARC Cores spun out of Argonaut Games and one of its successor companies, Argonaut Technologies, Limited, in 1996 as a commercial company and was acquired by Virage Logic in 2009. Virage Logic was acquired by Synopsys in 2010; thus, by 2014, Synopsys had three ADL-based technologies: nML (Target Compilers), LISA (LISATek), and ARC APEX (ARC/Virage).

The Second Industrial Era: 2010–2020

In this phase of ADL evolution, there have been some new ADLs emerging from industry. However, these did not divert much from the basic ADL semantics already established by the first- and second-generation ADLs.

During the second industrial era, Tensilica, which had started in 1997, was acquired by Cadence in 2013, and the decade saw further development of Tensilica configurable, extensible processor technology under the Xtensa name, including its ADL TIE. Synopsys, which by 2010 had two ADL-based technologies and added nML by 2014, continued to develop ARC APEX as part of its ARC offering. It also merged aspects of its ASIP technologies into a more unified tool called ASIP Designer (Synopsys ASIP Designer), which is the current home of nML within Synopsys today.

Because the fundamental ADL research had been done during the first and second generation of research and development, interest in ADL-based research in academia declined considerably during the second industrial era. Most academic interest lay in using ADLs to create interesting applications, as will be seen later.

Types and Characteristics of ADLs

Types of ADLs

Looking at ADLs in 2022, one can distinguish two basic types of ADLs: complete and ISA extension.

Complete ADLs allow the designer to capture all aspects of a processor and its ISA: all processor resources and properties and all aspects of the complete ISA including all basic scalar operations and all operation extensions including vector/SIMD operations and their associated resources and properties. A tool flow supporting a complete ADL will compile it into all relevant derivative design files needed to support the HW implementation, SW creation, and the HW-SW interface. The net result is an ASIP designed from the ground up. Examples of complete ADLs include MIMOLA, nML, EXPRESSION, and LISA.

ISA extension ADLs are intended to allow designers to extend a configurable, extensible processor by adding new operations/instructions to a basic core ISA. Somewhat ironically, the core ISA itself may also be partially captured using the ADL, but there is usually a “hard core” part of the ISA captured in configurable RTL and supported deeply within the SW tools. The extension ISA adds relevant resources, properties, and behavioral and structural descriptions of the additional instruction extensions, and these are compiled by the tool flow into additional HW implementations and SW tool properties that add to the basic hard core ISA support in the tool flow. The net result is an ASIP designed via additions to the core ISA.

Examples of ISA extension ADLs include Cadence/Tensilica TIE, Synopsys ARC APEX, ANDES ACE, and Codasip CodAL.

Characteristics of ADLs

In order to specify all aspects of a processor, whether used as a complete ADL or an ISA extension ADL, several different specification types need to be supported. Some of these may be supported in configurable RTL rather than directly in the ADL. Some of these may be specified in the ADL and used to generate configurable RTL (especially for structural hardware characteristics and instruction implementation details for optimal physical design). Some may be captured as forms/checkboxes/drop-down menu selections on a graphical user interface, while others are captured in a language-like syntax. The world of ADLs has over the past 30 years seen all these varieties used, and it has been up to researchers and commercial developers to decide on the variety of specific capture mechanisms used and supported.

Nevertheless, the following list of characteristics for the processor needs to be specified either directly in the ADL or by complementary specification methods:

- Endianness (big or little). Big-endian machines have declined in popularity but some ADLs and tool flows may still support them.
- Register file sizes (width, depth) and characteristics. There may be multiple register files, for example, scalar and vector registers and special accumulator registers including wide accumulators.
- Pipeline details, including length of pipeline, details of instruction fetch, decoder, bypassing, and allocation of operations to functional units. There may be separate scalar and vector pipelines and need to be defined linkages between them.
- All functional units, scalar and vector/SIMD, special functional units such as multiply-accumulate, floating point units (half, single, double precision), DSP instruction units with special capabilities such as fixed-point arithmetic, saturation, rounding, select/shuffle operand units, and special mathematical units (e.g., for divide, square root, transcendental functions).
- Memory interfaces – local caches at several levels and their characteristics; local tightly coupled memories; specialized instruction and data memories may be supported or a unified memory used.
- System-memory interfaces including higher level caches and memories, which may be accessed via standard buses.
- Bus interfaces – usually standardized, such as ARM APB/AHB/AXI, but sometimes proprietary, often with aspects of configurability even if standardized.
- Debug, tracing (instruction and/or data), and JTAG ports.
- Timers, interrupts, and exception vectors and configuration.
- Definition of VLIW-style or other flexible instructions (which may use multiple formats of different sizes to allow more efficient encoding than classical VLIW), which can pack and issue multiple operations in a single cycle, via explicit slotting of operations into slots.
  - This can duplicate functional units when the same operations or a subset (scalar, vector, SIMD) are desired to be in multiple issue slots for greater performance on various applications.

- Possible superscalar instruction assignment to functional units either in-order or with more complex logic, out of order.
- One or more load-store units.
- Definition of port, queue, and lookup style interfaces of arbitrary widths with access operations that link to the datapath.
- Special interfaces to ancillary hardware processing units.
- Windowing schemes for registers and function calling may be configurable.
- For each instruction, the characteristics are as follows:
  - Binary opcode and operand encoding (unless automatically generated by the tool flow), assembly syntax, C-syntax, use of registers, memory access capabilities for load/store operations.
  - Fundamental datatypes for the instruction (8-, 16-, 32- and 64- bit are common with 2- and 4-bit of interest in AI/ML). Sometimes very specialized datatypes, such as 12-bits, may be optimal for specialized operations.
  - Automatic compiler optimized use such as automated inference, vectorization.
  - Specific mapping to functional units, functional implementation, optimized hardware implementations including multicycle operation and schedules, resource sharing, specialized hardware unit sharing, and pipeline interlocking.
  - For SIMD style operations, the number of parallel lanes (2-, 4-, 8-,16-, 32-, and even 64- and 128-way are common with vector sizes up to 512 or 1024 bits, dependent on datatypes)
- Choice among various Application Binary Interfaces (ABIs) for optimized application development and compatibility across various ADLs and with intermediate software stacks.

Key ADLs

MIMOLA

MIMOLA is one of the oldest ADLs, predating the "classical" 1990s period by more than a decade (Marwedel 1979) and with continued development for two decades (Leupers and Marwedel 1998). It was developed at the University of Dortmund, Germany, and originally proposed for microarchitecture design. As befits the ADL concept, with MIMOLA the same description can be used for synthesis, simulation, test generation, and compilation. Its tool chain included the MSSH hardware synthesizer, the MSSQ code generator, the MSST self-test program compiler, the MSSB functional simulator, and the MSSU RT-level simulator, and MIMOLA has also been used by the RECORD (Leupers and Marwedel 1998) compiler.

The MIMOLA description is in three parts: the algorithm to be compiled, the target processor model, and additional linkage and transformation rules. The software part is an algorithm description using a PASCAL-like syntax for application programs. The target processor model uses a component netlist to define a microarchitecture. The compiler uses the "linkage" information to define important modules such as program counter and instruction memory.

The algorithmic part of MIMOLA is an extension of PASCAL; but unlike other high-level languages, it allows references to physical registers and memories and allows use of hardware components by using procedure calls.

In addition to using a netlist of component modules for the microarchitecture. MIMOLA permits modeling of arbitrary (programmable or nonprogrammable) hardware structures, and a library of predefined, primitive operators is provided. The basic entities of MIMOLA hardware models are modules and connections, where modules are specified by port interface and behavior. The MSSQ code generator extracts instruction-set information from the module netlist. Due to the generality of an RTL structural description, it can be tricky to extract a Connection Operation Graph (COG) and I-trees, even when the linkage information is provided. Thus, constraints need to be imposed in order for the MSSQ code generator to work properly: These constraints limit the architecture scope of MSSQ to microprogrammable controllers, in which all control signals originate directly from the instruction word. Since MIMOLA lacks an explicit description of processor pipelines or resource conflicts, this may result in poor code quality for some classes of VLIW or deeply pipelined processors.

The difficulty of instruction-set extraction can be avoided by abstracting behavioral information from the structural details; some behavioral ADLs, such as nML (Freericks 1993) and ISDL (Hadjiyiannis et al. 1997), explicitly specify the instruction semantics. These place less emphasis on the microarchitectural details.

EXPRESSION

A different approach was used in EXPRESSION , developed at the University of California, Irvine (Halambi et al. 1999), which describes a processor as a netlist of units and storage and automatically generates RTL based on the netlist. This is of higher-level abstraction, closer to a block diagram level description of an architecture.

EXPRESSION as an ADL has been used by various tools including a retargetable compiler and a simulator (Khare et al. 1999). The language has two main parts: a behavioral section with operation, instruction, and mapping descriptions; and a structural section with components, pipeline/data-transfer paths, and the memory system.

nML

nML was created at the Technical University of Berlin, Germany (Freericks 1993), and originally supported software tool generation from behavioral descriptions, more than hardware generation. However, over its long history as enumerated above, its capabilities were extended by Target Compiler Technologies (Goossens et al. 2006), and in the form of the Synopsys ASIP designer (Synopsys ASIP Designer), where it eventually ended up, it is a large part of a complete and sophisticated commercial ASIP generation system.

In its early days, nML was used by code generators CBC (Fauth and Knoll 1993) and CHESS (Lanneer et al. 1995) and instruction-set simulators – what was called CHECKERS (Goossens et al. 2006). The CHESS/CHECKERS environment was used for automatic and efficient software compilation and instruction-set simulation, but was extended to support HDL generation and test program generation (Goossens et al. 2006).

Very early in ADL history, the nML developers recognized the fact that several instructions share common properties, which could help make the final nML description compact and simple. Therefore, nML uses a hierarchical scheme to describe instruction sets. The instructions are the topmost elements in the hierarchy. The intermediate elements of the hierarchy are partial instructions (PI). Two composition rules, AND-rule and OR-rule, are used to establish relationships between elements. The AND-rule groups several PIs into a larger PI and the OR-rule enumerates a set of alternatives for one PI. Since instruction definitions in nML are thus in the form of an and-or tree, each possible traversal from the root to the leaf node of the tree gives an actual instruction.

Early nML also captures the structural information used by the ISA. For example, storage units are declared since they are visible to the instruction-set. nML supported three types of storage: RAM, register, and transitory storage. Transitory storage refers to machine states that are retained only for a limited number of cycles. Computations had no delay in the nML timing model – only storage units have delay. Instruction delay slots are modeled by introducing storage units as pipeline registers. The result of the computation is propagated through the registers in the behavior specification.

Early nML usage had a number of limitations that made it difficult to model complicated constraints found in DSPs with irregular instruction level parallelism or VLIW processing with multiple issue slots. A detailed review of nML from 1997 (Hartoog et al. 1997) discusses some of those limitations.

However, in its use by Target Compiler Technologies and then by Synopsys as a fundamental technology for ASIP design, its capabilities were extended or used in a complementary fashion with other aspects of ASIP designer to allow a wide variety of ASIPs to be designed with various architectures. A good overview of nML as currently used in ASIP Designer is available in (Bo and Willems 2015).

LISA

Language for Instruction Set Architecture (LISA) (Meyr et al. 2008) was developed at RWTH Aachen University, Germany, originally to help in developing fast simulators (Nohl et al. 2002). By trading off speed and accuracy constraints, different modes of instruction-set simulator, for example, compiled, interpretive, and just-in-time cache-compiled (JIT-CC), could be generated. LISA could be used in a stepwise fashion to gradually increase the level of details: A designer could start with an instruction-accurate LISA description, carry out early design space exploration, and then refine the design to a detailed, cycle-accurate model. To support this kind of design, application profiling, automatic instruction-set encoding

(Nohl et al. 2003), as well as custom instruction identification (Leupers et al. 2006) played an important role. From a cycle-accurate LISA description, optimized, low-power RTL (Chattopadhyay et al. 2006a; Chattopadhyay et al. 2006b) generation was possible. LISA also provided a methodology for automated test pattern and assertion generation (Chattopadhyay et al. 2006c) and was also used to generate retargetable C compilers (Hohenauer et al. 2004; Wahlen et al. 2003).

During the LISATek era, the language was extended to cover a wide range of processor architectures such as VLIW, weakly programmable ASICs (Wang et al. 2012), Coarse-Grained Reconfigurable Architectures (CGRAs) (Chattopadhyay et al. 2008), and partially reconfigurable ASIPs (rASIPs) (Chattopadhyay et al. 2009). However, after acquisition by Synopsys, and its further acquisition of Target Compiler Technologies with nML in 2014, specific aspects of LISA technology were subsumed into the ASIP Designer toolset, which is heavily based on nML, and any specific LISA-based capabilities are no longer easy to find in ASIP Designer.

PEAS

PEAS , which went through several generations (PEAS-I, PEAS-II, and PEAS-III) in the 1990s (Itoh et al. 2000a; Itoh et al. 2000b), was somewhat unique in being an ASIP approach that emerged from academic work in Japan. It was the basis for a commercial spin out, ASIP Meister (Hassan and Imai 2005), but this was not very successful. In addition, although the PEAS developed an ASIP design environment, it was more of a GUI-based environment that did not share much in the way of ADL formalisms.

TENSILICA TIE

To manage the complexity of processor design, configurable, extensible processor cores were provided by Tensilica, which started out at the end of the 1990s as a commercial company and was acquired by Cadence's IP division in 2013 (Cadence Tensilica). Although Tensilica Xtensa restricts the ASIP design space to some extent by providing a base RISC ISA and architecture, the capabilities of TIE allow a very wide range of ASIPs to be generated. Thus, one can classify TIE as an ISA extension ADL using the simple categorization above, rather than a complete ADL.

For Tensilica's Xtensa configurable, extensible cores, the configurable base Xtensa RISC is a design space that is configured by discrete choices such as basic microarchitecture (there are two choices, LX and NX), number of pipeline stages (LX offers 5 and 7 stage), depth of the basic address register file, windowing, cache and tightly coupled local memories, and many other parameters as described in (Augustine et al. 2009). On the other hand, users can model, for example, arbitrary single-cycle or multicycle custom instruction-set extensions, register files, VLIW formats, vectorization rules with an ADL, known as Tensilica Instruction Extension (TIE). TIE instruction extensions are added to an Xtensa processor instruction pipeline, which typically contains a 5-stage or a 7-stage pipeline (LX) or 10-stage

pipeline (NX). An example TIE description for a 4-way, 16-bit vector integer add instruction is shown in the following:

```
regfile simd64 64 16 v//16-entry register file
that is 64 bits wide operation vec4_add16
{out simd64 res, in simd64 A, in simd64 B} { } {
  wire [15:0] rtmp1 = A[15: 0] + B[15: 0] ;
  wire [15:0] rtmp2 = A[31:16] + B[31:16] ;
  wire [15:0] rtmp3 = A[47:32] + B[47:32] ;
  wire [15:0] rtmp4 = A[63:48] + B[63:48] ;
  assign res = { rtmp4, rtmp3, rtmp2, rtmp1 } ;
 }
```

Following the automatic compilation of TIE by the Xtensa TIE compiler, a number of artifacts are generated, including design files which extend the C/C++ compiler, the ISS, and other software tools, as well as HDL descriptions that can be used to build the configured, extended processor. This allows the example instruction *vec_add16* to be used in the application C-code as a C intrinsic call, as below:

```
simd64 A[VECLEN/4], // Input vectors simd64 B[VECLEN/4],
// Input vectors simd64 sum[VECLEN/4]; // Output vectors for
(i=0; i<VECLEN/4; i++) {
 sum[i] = vec4_add16(A[i],B[i]);
}
```

More detailed descriptions of TIE are available at Sanghavi and Andrews (2008) and (Bailey and Martin 2010, chapter 6).

The design space exploration flows supported in the Xtensa and TIE toolset present opportunities for exploiting data-level parallelism, instruction-level parallelism using Xtensa Flexible Length Instruction eXtensions (FLIX, which is similar to VLIW but with better encoding opportunities since it supports multiple formats), customizable storage, and increased data bandwidth. For example, to have increased data bandwidth, processor designers can add multiple I/O interfaces (ports, queues, lookups) to the Xtensa processor for fixed-latency data transfer. Instructions created using TIE implicitly initiate one or multiple transactions over these interfaces, which significantly increases I/O bandwidth. These can also be used to interface to external dedicated hardware blocks which execute autonomously from the basic Xtensa pipeline.

ARC APEX

ARC promoted the concept of configurable processor development, where a basic microarchitecture could be customized and extended with the help of a Graphical User Interface (GUI). The configuration tool, named ARChitect, allowed users to select preconfigured options and design custom instructions on top of the 16/32-bit ISA. Typical configuration options include types/sizes of core registers, address widths, and choice of instruction sets.

The modified architecture is also supported by automatic generation of synthesizable RTL in addition to software tools such as testbench, compiler, debugger, simulator, and documentations.

Since Synopsys acquired Virage in 2010 (which had acquired ARC in 2009), APEX has been an important, if not extensively used, aspect of the ARC cores (EM, HS and VPX/EVX families). Synopsys has used it for some of its own packages of instruction extensions. It does not provide much external information on it, although a bit can be gleaned from a white paper (Geuzebroek 2014) and an academic research paper (Ogawa et al. 2019). It is clear that ARC APEX is an ISA extension ADL, not a complete ADL, and its abstraction level for operations is close to the Verilog HDL level with a few extra capabilities to describe the operations. These are then used as intrinsics within ARC C-code.

Codasip CodAL

Codasip (Codasip) emerged during the third ADL era (the second industrial era) as a commercial ASIP provider, with its own proprietary ASIP RISC cores, proprietary ISA extension ADL CodAL (Codasip Architectural Language) (Přikryl 2020), and GUI and toolset for defining and building ASIPs. Like many earlier ADLs, it was based on academic work – in this case, done at the University of Brno, Czechia (Trmac et al. 2010; Husár et al. 2010).

However, it was not successful with its proprietary approach, so it cleverly did a pivot in the middle of the era to switch out its proprietary RISC for RISC-V, which was emerging as a new standard, defining a new family of RISC cores based on various RISC-V extension bundles, and adapting CodAL to generate implementations of new instruction extensions to fit into RISC-V architectural concepts.

In late 2021–early 2022, Codasip also began significant management changes and expansion of R&D beyond its Brno, Czechia roots. (Press Release 2021). It is not clear if it will be successful long term as an offering of RISC-V base cores, which can be extended by users using CodAL, but the pivot was clearly an important part of corporate survival.

Codasip offers Codasip Studio (https://codasip.com/products/codasip-studio/) as a GUI and toolset to convert base RISC-V configurations plus CodAL ISA extensions into a buildable ASIP with software tool support. The following code example shows a Multiply-Accumulate (MAC) instruction realized using CodAL, which draws inspiration from ADLs like LISA.

```
element i_mac {
use reg as dst, src1, src2;
assembly {"mac" dst "," src1 "," src2};
binary {OP_MAC dst src1 src2 0:bit[9]};
semantics {rf[dst] += rf[src1] * rf[src2]};
}
```

Andes ACE

Andes (http://www.andestech.com/en/homepage/) is another IP company with long roots in Taiwan, originally offering CPUs and DSPs built on its own proprietary RISC ISA. It offered an ISA extension ADL, Andes Custom Extension (ACE) (Andes Custom Extension) in a manner similar to ARC APEX, Tensilica TIE, and Codasip CodAL.

Despite reasonable success in Taiwan with its proprietary CPUs and DSPs, Andes realized that it could not break out to become a more worldwide IP company using that approach. Therefore, in a manner similar to Codasip, Andes did a pivot during the second industrial era to switch from its proprietary RISC to RISC-V and to evolve all its product offerings to focus on RISC-V in the future. While still offering the legacy RISC-based CPUs and DSPs, all work in recent years has been based on RISC-V.

This includes all development work on ACE (RISC-V Custom 2021). As an ISA extension ADL, it is not particularly innovative, and its success in the RISC-V domain (as compared, for example, to Codasip) will no doubt rest on the underlying corporate capabilities and base CPU and DSP designs, more than on the ADL technology.

RISC-V Chisel

In a different approach, Chisel (Chisel Programming Language) was proposed to describe circuits with much more brevity and intuition compared to the traditional RTL description languages. The key motivation for Chisel is to utilize the power of object-oriented languages and functional programming, for which the Scala programming language is chosen as the basis. Modules and datatypes can be described as objects, from which various components can be inherited with strong type enforcement. From Chisel, automatic generation of synthesizable RTL descriptions is supported. While Chisel does not support generation of software tool-suites like previously described ADLs and also is not restricted specifically to programmable processors, it is certainly an interesting addition to the class of abstract hardware description languages and therefore is worthwhile noting here.

A new entrant in the domain of ADLs, Chisel had the advantage to study the merits and demerits of prior approaches. However, even with the examples of complete ADLs and ISA-extensible ADLs and the long history of both, the developers of Chisel restricted their scope to really be a higher-level hardware language or "higher-level RTL."

Its use for developing general hardware IP beyond the RISC-V programmable processors has been noted, but most attention is paid to RISC-V processors, where it faces limitations. Although some RISC-V suppliers (SiFive (https://www.sifive.com) being most prominent) have continued to base their IP developments on Chisel, it is interesting to note that others such as Andes and Codasip use a

combination of conventional configurable RTL design for their basic CPUs and DSPs, and their own ISA extension ADLs for adding new custom operations, rather than anything based on Chisel.

ADL-Driven Methodologies

There are five areas of particular interest where ADL methodologies have made major contributions in both academic research and the commercial domain:

- Generation of software tools
- Automatic synthesis of custom instructions for an application
- Instruction-set simulator generation
- Generation of optimized hardware
- Verification

Generation of Software Tools

As discussed previously, ADLs are used to specify processor and memory architectures and have been used to generate software tools including the compiler, simulator, assembler, profiler, debugger, disassembler, hardware abstraction level software (HAL), often software stacks and configured low-level software routines, and libraries tuned to the processor configuration. Figure 2 shows an ADL-based design space exploration flow. The application programs, usually written in C/C++ or in higher level libraries such as OpenCV, OpenCL, or Halide, are compiled to the ISA and simulated, and the feedback is used to modify the ADL specification, to explore the design space, with the goal of finding the best possible architecture for the given set of application programs under various design constraints such as performance, power and area (PPA).

Drawn from the three eras of ADL development, there are many detailed descriptions of software tool generation and associated design space exploration. These include ISDL (Hadjiyiannis et al. 1997), Valen-C (Inoue et al. 1998), MIMOLA (Leupers and Marwedel 1998), LISA (Meyr et al. 2008), nML (Freericks 1993), Sim-nML (Rajesh and Moona 1999), EXPRESSION (Halambi et al. 1999), Synopsys ARC (Synopsys DesignWare ARC), RADL (Siska 1998), Synopsys ASIP designer based on Target Compiler Technologies (Synopsys ASIP Designer), Tensilica TIE (Cadence Tensilica), MDES (The MDES User Manual), Codasip Studio, and ANDES.

Compilers Traditionally, software for embedded systems was hand-tuned in assembly. However, it is not practical to develop software in assembly language or to optimize it manually except for critical sections of the code. The use of intrinsics embedded in C/C++ code can substitute for manual assembly to some extent and is often as efficient, but in general, high-quality compilers which produce optimized machine-specific code from a program specified in a high-level

language (HLL) such as C/C++ and Java are necessary for productivity, time-budgets, and efficiency, in order to produce efficient software within the time budget. There has been a lot of work on efficient compilers for embedded systems (Hohenauer et al. 2006, 2008; Goossens et al. 2006). In particular, given the rise of ASIPs, new processor ISAs such as RISC-V, application-specific processing in general, and new compilation approaches such as CLANG/LLVM, the need to make all good compilers retargetable has come to the fore (Goossens et al. 2006).

Automatic Synthesis of Custom Instructions for an Application

Use of ISA extension ADLs to complement a base ISA, as with ARC APEX, Tensilica TIE, Codasip CodAL, and ANDES ACE, begs the question of where the new custom instructions will come from.

Skilled designers, using the ADL ASIP tool capabilities and application profiling, are usually a good source of new custom instructions. However, in the early to late 2000s, there was considerable interest in automating this process. In (Atasu et al. 2012; Pothineni et al. 2008; Biswas et al. 2007), the problem was looked at in a standalone fashion. Custom instruction, as a standalone problem, has been studied in depth (Atasu et al. 2012; Pothineni et al. 2008; Biswas et al. 2007). For ISA extension to an existing processor, there have been research work (Leupers et al. 2006) and work on hardware optimizations (Karuri et al. 2007) and with reconfigurable targets (Karuri et al. 2008).

Perhaps more interesting are two developments that tried to commercialize this approach. The Tensilica XPRES tool [(Goodwin and Pekov 2003), Chapter 8 of reference 2, chapter 6 of reference 84] automatically moved from a single or multiple application profiling runs to the generation of Tensilica TIE ADL code extending operations in three ways: SIMD, instruction fusion, and instruction-level parallelism (FLIX). The automatically generated TIE could be constrained to land at different points along a performance-area Pareto curve, thereby facilitating design tradeoffs. Equally interesting, the compiler could automatically make use of the automatically generated instructions to speed up the code when compiling it, without manual intervention, thus providing a high level of automation.

Another attempt was based on the work of Pozzi and Ienne [Chapter 7 of reference 84], which formed the basis of a startup in Lugano, Switzerland, called Mimosys (Brown and Epalza 2006).

Perhaps the most interesting conclusion from the Tensilica XPRES experience was that despite generating potentially thousands of lines of ADL code for instructions that gave significant acceleration for a class of applications, the resulting code was not really the basis for a commercial product by a design team. It was too lengthy and complex, was not commented (so the links back to the kernels of the applications were difficult to discern), and lacked enough generality to be a good basis for future applications drawn from the domains used to generate the ADL code.

Instruction-Set Simulator Generation

Instruction-set simulators (ISSs) are essential components for the processor designer. They are used for many tasks: verifying the functionality, timing behavior of the system (including hardware and software), and generating measurements (e.g., power consumption (Xie et al. 2013)) which can aid design space exploration. Thus, ISS forms an integral part of virtual prototyping flows (discussed in depth in ▸ Chap. 27, "Virtual Prototyping of Processor-Based Platforms").

There is a tradeoff between the abstraction level of an ISS and the runtime performance and accuracy of the simulation. The highest level of abstraction is a functional simulation of the processor instruction set those models only the instruction behavior, without regard for cycle-time characteristics. This is an instruction-accurate (IA) simulator. As abstraction descends to lower levels, there are cycle-accurate and phase-accurate simulators that give more detailed timing information. Various other aspects of ISSs include whether they provide bit-accurate models, pin-accurate models, exact pipeline models, and structural models of the processor.

Higher-level ISSs run faster, but generate less information than lower-level ISSs. Since ASIPS and a variety of ISA targets are important in this methodology, having a retargetable simulator base that can generate ISSs including ISA extension models for a large variety of targets is very desirable. In addition, a capability of generating several types of ISSs: functional (instruction accurate), cycle-approximate (with general modeling of the pipeline), cycle-accurate (with accurate pipeline modeling), transaction-accurate (using faster, more abstract data structures for passing control and data information into and out of the simulator interfaces), and pin-accurate (precise modeling of all interfaces at the pin/signal level) makes the ISS generation framework very flexible.

The academic world has worked on a variety of general ISS frameworks such as QEMU (https://www.qemu.org) and also target-specific ISSs such as Spike for RISC-V. Each ADL project in general, whether academic or commercial, will come with its own ISS generation capability. Over the years, there has been a shift in the underlying ISS technology: from interpreted simulation, through compiled simulation, through to what is called "Just in time Compiled Code simulation" which is a unification of the two methods (Nohl et al. 2002). In addition, hybrid simulation technology allows switching of abstraction levels to offer speed where accuracy is less important and accuracy where speed is less important – sometimes via static switching and sometimes dynamically.

Interpreted. An interpreted ISS uses a high-level model of the processor and its ISA. The state of the processor is stored in memory, and the simulator uses a fetch, decode, and execute cycle on target software in memory. The target software is in binary form and thus needs decoding, and then the various operations are executed serially, using the model for each operation.

An interpreted ISS is easy to implement, but has a high overhead compared to other approaches because of its fundamental interpretive nature. It can model some

of the internal pipeline and processor state and thus produce cycle-approximate statistics or, with a very detailed pipeline and state model, cycle-accurate statistics. There may be some difficulties accurately modeling instruction-level parallelism and the interaction of external events such as interrupts and exceptions since fundamentally every operation is executed serially, even if modeled as executing in parallel. Similarly, modeling the implications of various memory accesses accurately will require a lot of simulation overhead, thus slowing it down.

As will be seen, there are other approaches that tend to avoid the high overhead of the interpreted ISS.

Compiled. In a compiled approach, each target instruction is translated into a series of host machine instructions which emulate the function of each target instruction and use and modify the simulated processor state.

If the target code is compiled into host code before running the simulator, the complete processor fetch-decode-dispatch overhead is eliminated (this is static compiled simulation). Alternative approaches will compile the code dynamically when it is loaded; in this way, the overhead will be spread over repeated executions.

Compiled ISSs have been the subject of research (Zhu and Gajski 1999; Pees et al. 2000; Cmelik and Keppel 1994; Witchel and Rosenblum 1996) and commercial ASIP/ADL providers have also offered them. Synopsys ARC, for example, offers nSIM, which is based on compiled ISS concepts, to complement its xCAM Cycle-Accurate ISS model, which is generated by abstracting from the processor RTL (https://www.synopsys.com/designware-ip/processor-solutions/arc-development-tools/simulation-tools.html).

Interpreted and Compiled. Because interpreted ISSs are flexible but slow due to instruction decoding and dispatch modeling, and compiled ISSs impose a preruntime overhead to generate the static compiled code simulation model, researchers worked on methods to combine the advantages of both, such as "Just in Time Compiled Code Simulation (JIT-CCS)" (Nohl et al. 2002) and "Instruction Set Compiled Simulation (IS-CS)" (Reshadi et al. 2003).

In JIT-CCS, an instruction is compiled during runtime, just before the operation needs to be executed, and the compiled information is cached so that when the operation is encountered again in the instruction stream, it does not need to be recompiled. In IS-CS, instruction decoding is done during compile time, and if the code is run-time modified, the instruction is re-decoded in subsequent use. To further reduce the overhead of the compilation process, the ISS framework might execute an operation a few times in interpreted mode and only compile it once it has detected that the operation is likely to be executed many times. Thus, very infrequently executed instructions (or indeed, instructions that may not be executed at all in a particular simulation run) are only interpreted and not compiled.

These techniques did not remain as only of academic research. Commercial ASIP/ADL companies adopted these techniques to speed up their ISS simulation. Tensilica, for example, introduced a "fast functional" simulation mode called TurboXim (Augustine et al. 2009), which complements its cycle-accurate ISS mode, and greatly increased functional ISS performance for software developers.

Hybrid. Since software developers often do not need high accuracy in simulation for a complete application run, hybrid simulation offers a capability to use functional execution to quickly reach an area of interest and then switch to a detailed cycle-accurate simulation mode to debug or study the detailed performance of a critical area of code. Researchers developed such techniques (Kraemer et al. 2007) allowing switching between cycle-accurate modeling and functional mode which uses host-based emulation.

The main issue with hybrid simulation is keeping the models of the processor state in the two modes synchronized enough that switching does not introduce gross inaccuracies. This can be done by restricting where switching is done (e.g., at the boundaries of a function) or by explicit state flushing and restoration techniques done just before switching,

The hybrid technique is also available in the commercial ASIP world (see, for example, Chap. 6 of Bailey and Martin 2010), where it can also be used in a different mode – statistical sampling between fast functional and cycle-accurate mode, to allow performance predictions based on cycle-accurate execution to be generated for the complete application running mostly in fast functional mode. As with any technique, care must be taken to ensure that the predictions have reasonable accuracy by ensuring that mode-switching is sufficiently random.

Generation of Hardware Implementation

Detailed PPA analysis of generated processors, and integration into an actual SoC, needs synthesizable RTL. The two types of ADLs discussed earlier: complete ADL and ISA extension ADLs will do this in somewhat different ways. In the end, however, much of the PPA of the generated processor depends on the degree to which the designers take care to optimize the resulting core for their applications.

ISA-Extensible ADL In this approach, followed by most commercial ASIP/ADL providers (e.g., Tensilica TIE, Synopsys ARC, Andes ACE, Codasip CodAL), a base core architecture is configured through the selection or de-selection of many different offered functional units, ISA packages, interfaces such as memories, pipeline choices, etc., as discussed earlier. Complementing this configuration is the designer addition of ADL-based ISA extensions, which as noted can contain considerable extra processor resources (register files, vector SIMD units, VLIW style slotting of duplicate functional units, etc.). Thus, in this approach, the base processor once configured may not be extended at all, or may be modestly extended with a few simple operations, or may be vastly extended with complete vector SIMD processing units, so that the resulting generated HDL could be 100% based on the supplier design, or anywhere from a few percent to 90% or more based on designer ISA extensions. This is the basis of a "configurable, extensible" processor offering.

As the percentage of customized ISA extensions rises, the difference between an ISA-extensible ADL and a complete ADL begins to diminish. Nevertheless, as

will be seen, complete ADL approaches offer yet more flexibility in basic processor architecture.

Under the hood, a processor supplied ASIP/DSP may actually use its own ISA extension ADL to generate much of the base processor configured RTL code. The reason for this being the high-quality RTL code generation from an ADL. In this scenario, libraries of preoptimized ADL code may be offered as part of the toolset to give designers a quicker path to efficient RTL and more optimal RTL than the external designs could achieve on their own. This follows the mantra that "the best designers of ADL-based processors are often the internal supplier designers."

Complete ADL Researchers and providers of complete ADL ASIP approaches have offered more flexibility with this approach than the ISA extension ADL concepts discussed previously, although, as mentioned, when user-defined ISA extensions grow to be 90% of the processor, the distinction between the two approaches is greatly diminished.

Nevertheless, complete ADL approaches offer designers more flexibility in bottom-up ASIP design, because they do not restrict them to a preimplemented pipeline and basic processor micro-architecture. Nor do they impose a predefined base RISC ISA. Tools such as Synopsys ASIP designer (Synopsys ASIP Designer) offer the designers a chance to completely define the pipeline, interfaces, and ISA for the generated processor. In many deeply embedded applications (such as hearing aids or audio processors), this ability to completely tune the resulting PPA of the generated HDL may be a vital design criterion that outweighs the extra design time needed to start from scratch.

Commercial providers of complete ADLs try to compensate for the steeper learning curve when compared to ISA extension ADL approaches by providing extensive example libraries. These give starting points in source code for users to learn processor design from the ground up and to develop their own ground-up designs. Synopsys ASIP designer, for example, provides examples ranging from simple 16-bit RISC microcontrollers, through 32-bit microcontrollers, RISC-V CPUs, DSPs from simple through SIMD through VLIW style, and a number of ASIP designs for video, communications, FFT, JPEG encoding, image processing, matrix operations, and Simultaneous Localization and Mapping (SLAM) (Synopsys ASIP Designer).

Top-Down Verification

Verification of processors is one of the most important tasks in SoC design. Traditionally, a verification methodology for processors would start from a high-level architectural specification and attempt to ensure that the real implementation matches the specification using a variety of techniques ranging from intensive simulation to some level of formal verification. Since specifications were usually natural language, the effectiveness of the verification rested on the correct translation of design intent into simulation testcases and the completeness of simulation test coverage.

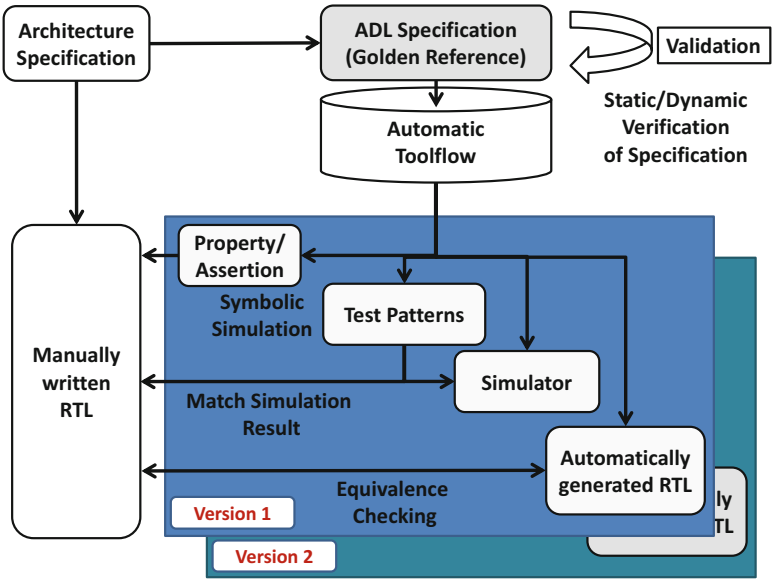


Fig. 3 Top-down verification flow

With ADLs, designers were able to develop *executable specifications*, either for the complete processor or for a major portion of it, and this allows a top-down verification flow to be developed (Mishra 2005; Chattopadhyay et al. 2006c) as depicted in Fig. 3.

Verification based on ADLs may use two concepts: first, the ADL specification of the processor is validated for completeness and integrity using a variety of methods, and second, the ADL specification can be used to drive simulation-based verification processes.

Validation of an ADL Specification

Because the ADL specification is compiled by tools in ADL-driven design flows, and artifacts are automatically derived from the output of a successful compilation, a certain amount of basic ADL specification integrity is checked during this compilation process. For example, properties of instructions that contradict each other due to errors or typos will be normally flagged. Specifications for multicycle operation schedules that are inconsistent with the implementation may become errors. Portions of an operation implementation that are declared but not used may trigger warnings. Inconsistencies in bit-widths in assignments may be warned against or trigger errors. What compiles successfully in the ADL specification should thus have basic integrity.

Beyond basic specification integrity, it is essential to verify static and dynamic properties and behavior. Static properties, for example, connectedness, false pipeline and data-transfer paths, and completeness, can be validated by using a graph-based model of the pipelined architecture (Mishra and Dutt 2004a). Dynamic behavior may be validated by analyzing the instruction flow in the pipeline. This could use a Finite State Machine (FSM)-based model to verify important architectural properties; for example, determinism and in-order execution in the presence of hazards and multiple exceptions (Mishra et al. 2003).

Use of assertion-based methodologies will be easier if they can be generated automatically from the ADL specification. In (Chattopadhyay et al. 2006c), LISA-generated assertions may detect incorrect dynamic behavior: for example, multiple write access to the same storage element.

Formal equivalence checking is supported in some commercial ASIP flows to help validate ADL specifications. For example, with Tensilica TIE, it is possible to describe ISA extensions in two different ways: a functional description of individual operations using simple TIE constructs and a "semantic" description of a functional unit that may implement multiple operations using TIE constructs that are closer to Verilog HDL in abstraction level. From these two different descriptions, the TIE compiler can generate two different HDL representations for the operations under design and then use formal equivalence checking on the two representations to check that they truly implement the same functionality. This is part of a set of verification capabilities described in (Jani et al. 2005).

Specification-Driven, Simulation-Based, Verification

A validated ADL specification can be used as a golden model for top-down verification flows. Such a top-down verification flow has been used in several contexts: for example, functional test program generation, verification using equivalence checking, and symbolic simulation.

Test generation for functional simulation-based verification of processors was demonstrated in the early ADL eras using MIMOLA (Leupers and Marwedel 1998), EXPRESSION (Mishra and Dutt 2004b), LISA (Chattopadhyay et al. 2006c), and nML (Synopsys ASIP Designer). With EXPRESSION (Mishra and Dutt 2004b), a model checking approach was used to automatically generate functional test programs from the processor specification. This worked by generating a graph model of the pipelined processor from the ADL and then creating test programs to cover the behavior of the pipeline. Further work in this vein was reported for EXPRESSION (Dang et al. 2009) and LISA (Chattopadhyay et al. 2006c).

A slightly different verification approach using equivalence checking to that presented in the previous section was suggested in (Mishra 2005). Here an ADL-generated RTL model was compared to a hand-written RTL implementation of the processor. The approach also generated properties that could be checked with symbolic simulation.

Commercial design flows for ASIPs with ADLs generate many simulation-based verification artifacts automatically or based on libraries of models supported in the flow, and these can be useful for user-defined ISA extensions as well (Jani et al. 2005). However, one can hardly say at this point in 2022 that the verification problem has been solved, for hardware, software, or combinations thereof, despite continued attention to the issues by EDA companies and design teams. As with ADLs in general, academic research in formal verification has focused on new constraints, such as, security (Watson et al. 2019).

Applications of ADL-Based Design

The idea of ADLs is to create application-specific processors by empowering designers. Hence, it is worthwhile to take note of the various processors developed using the ADLs. Several such processors are summarized in Table 1, many of which have been deployed commercially. One of the design projects, where additional features are plugged in to an existing ADL, is described in a little more detail after the table, drawing out important lessons from using ADLs.

Tensilica TIE In (Martin and Nicolaescu 2018), a framework and advanced toolset and flow based on the Tensilica TIE ADL are described. This framework goes beyond the TIE ADL to add a number of capabilities that could be contemplated in future ADLs and tool flows and have been used to deliver various commercial DSPs by Cadence Tensilica. Much of this additional capability was developed using embedded Perl with TIE and associated Makefiles and processor building capabilities, but it is easy to see that these kinds of features could be directly added to ADLs themselves in the future.

The important capabilities added by this superset ADL flow include:

- Generation of families of similar DSPs, primarily by varying SIMD width for operations and register width for vector DSPs. Most vector operations operate in a SIMD fashion, so generating a 128-bit, 256-bit, 512-bit, and 1024-bit family of vector DSPs for the most part uses simple SIMD specification and lane-wise duplication. Some operations (such as select/shuffle with associated patterns) will need to differ by width in more complex ways, but even these can be described using common generators with SIMD variations embedded. The SIMD variation was called the "N-way" model.
- Complementing the N-way model for a family of SIMD-varying DSPs was an N-way programming model, where loops that only varied (in C/C++ programming) by SIMD width could be written using "N" symbolically for intrinsics for vector operations, and the number of passes needed for loops. With automated vectorization possible in the compiler, applications written in scalar C could be efficiently turned into highly loop-based assembly code that would work well across all SIMD variations in the DSP family. Even more complex operations such as select/shuffles could be made portable across SIMD widths

Table 1 Examples of ASIPs designed using ADLs

ADL	Image processing	Security	Audio	Comms	Others
ARC APEX	Filter (Bit 2020)	V2X (Ogawa et al. 2019)	Subsystem (van der Wolf and Derwig 2013)	IoT (Grimblatt et al. 2019)	Sensors (Geuzebroek 2014)
Tensilica TIE (Martin and Nicolaescu 2018)	Vision (Efland et al. 2016)	Root of Trust (Kalluri)	HiFi (Maydan 2011)	ConnX BBE (Rowen 2012)	KNN (Jamma et al. 2016)
LISA	Retinex ASIP (Saponara et al. 2007)	Crypto Pairing ASIP (Kammler et al. 2009)	Codec (Selim et al.)	FlexDet MIMO (Chen et al. 2015)	OSIP (Zhang et al. 2013; Castrillon et al. 2009) SVM (Gupta and Pal 2015)
nML	Motion Estimation (Machetti 2018)	SHA-3 (RAO and Jinli 2018)	CoolFlux (Roeven et al. 2004)	HSDPA (Rounioja and Puusaari 2006)	RISC-V (Cox and Taussig 2016)
Codasip CodAL	Face (Podivinsky et al. 2018)	AES (Podivinsky et al. 2018)	Codec (Podivinsky et al. 2018)	IoT Wireless (Amor et al. 2021)	RISC-V (Přikryl 2019)

by using symbolic (#define) pattern names and mappings (e.g., select every m-th. element from an input vector set).

- Many DSPs share many common subsets of operations used for basic vector operations (adds, subtracts, shifts, multiplies, multiply-accumulates). Additionally, some DSPs need complex datatypes and operations and some do not, while all of them need real arithmetic. Some DSPs need FIR and FFT operations, and some do not. By turning the definitions, properties, and implementations (in TIE ADL) for these subsets into reusable libraries of operations in various classes, many different types of DSPs could be created from a common option-based design flow built on top of these libraries. (This is something the RISC-V creators realized as well of course.)
- New ADL-based module definition capabilities could speed up the definition and generation of lane-wise SIMD operations by declaring processing for one SIMD lane and simply replicating it with an ADL construct. This avoided a number of errors that could occur when the SIMD lanes needed to be manually generated in the ADL (e.g., due to typos) and made the ADL code much easier to understand.
- Automated capabilities to define the verification requirements for each operation class and to generate verification tests from these higher-level requirements were introduced. Although this was done using tools and flows rather than in the ADL itself, it is easy to see (especially given earlier work) that this would be better added to the ADL as an additional set of representations or models.
- Finally, some fairly elaborate capabilities for documentation generation (e.g., HTML pages for every operation) were created. Many operations share common templates and developing documentation entirely by hand is inefficient and error-prone. The productivity increase by using automated flows was large.

The net impact of this kind of superset ADL flow was a further increase in design and verification productivity – and thus, this would be a good set of directions for future commercial ADP-based ASIP tools.

Conclusions

Heterogeneous multiprocessor SoCs and the rise of domain-specific computing have considerably increased the importance of application-specific processors and accelerators. It would have been impossible to design and verify the wide variety of ASIPs needed without the support of automated tools and methodologies: hence, in a multiera fashion, the emergence of ADLs from academic research and their increasing use in industry and as part of important commercial ASIP offerings. Diversions such as Chisel for some RISC-V aside, the ADL approach seems a vital part of current and future processor design no matter what ISAs prove most populator – and even if the world converged to just a few ISAs, because effective ISA extension demands an ADL approach.

It is interesting to note that in the third era: what has been dubbed the "second industrial era," academic research into ADLs themselves has declined considerably,

and most advances are the outgrowth of commercial entities. This is despite the fact that radically new processor architectures and design approaches are not well supported by ADLs. Perhaps academic research into new architectures will trigger new academic research into ADLs, especially in hot domains such as Machine Learning.

In future, the lessons learnt from the ADL evolution can prove to be beneficial for adopting new high-level synthesis technology trends. Considering the emergence of Machine Learning (ML) accelerators across various domains right now, it is only natural to think of whether there are effective design automation methodologies in place for automating the design of ML accelerators and what kind of parallels can be drawn from the ADL research. The development of ADL-based design flows coincided with the growth of wireless signal processing algorithms, which differ in many requirements compared to ML tasks, most prominently due to the aspect of data-driven computing of ML. Therefore, for such tasks that put emphasis on prudent memory bandwidth management and demand extreme forms of parallelism, newer forms of architecture and corresponding ADL-based tool flows can be definitely imagined.

The overview started with Makimoto's wave and the cycles of standardization and customization. Since domain-specific computing is one of the few techniques available to increase performance beyond the sputtering out of Moore's Law and Dennard scaling, it is entirely possible that domain-specific computing, with ADLs forming the important backbone, will break the wave and keep the industry in the customization cycle for a long time to come. The prediction of Chris Rowen, founder of Tensilica, that "the processor is the NAND gate of the future" (Gries and Keutzer 2005) may turn out to be the actual future, in no small part due to ADLs.

References

Amor HB, Bernier C, Prikryl Z (2021) A RISC-V ISA extension for ultra-low power IoT wireless signal processing. In: IEEE transactions on computers. Institute of Electrical and Electronics Engineers, pp 1–1. https://doi.org/10.1109/TC.2021.3063027.cea-03158876

Andes Custom Extension. http://www.andestech.com/en/productssolutions/andes-custom-extension/

Atasu K, Luk W, Mencer O, Ozturan C, Dundar G (2012) FISH: fast instruction synthesis for custom processors. IEEE Trans Very Large Scale Integr Syst 20(1):52–65

Augustine S, Gauthier M, Leibson S, Macliesh P, Martin G, Maydan D, Nedeljkovic N, Wilson B (2009) Chapter 7 Generation and use of an ASIP software tool chain. In: Wolfgang E, Wolfgang M, Rainer D (eds) Hardware-dependent software: principles and practice. Springer, Berlin/Heidelberg, Germany

Bailey B, Martin G (2010) ESL models and their application. Springer

Biswas P, Dutt ND, Pozzi L, Ienne P (2007) Introduction of architecturally visible storage in instruction set extensions. Comp Aided Design Integrated Circuits Syst IEEE Trans 26(3):435–446

Bit A (2020) 64-bit custom Math ISA in configurable 32-Bit RISC processor. In ICDSMLA 2019. Springer, Singapore, pp 564–575

Bo W, Willems M (2015) Rapid architectural exploration in designing application-specific processors. White Paper, Synopsys. https://www.synopsys.com/dw/doc.php/wp/architectural_-exploration_designing_application_specific_processors.pdf

Brown J, Epalza M. Automatically identifying and creating automatically identifying and creating accelerators directly from C code. Xilinx Xcelljournal, Issue 58, Third Quarter 2006. https://www.epfl.ch/labs/lap/wp-content/uploads/2018/05/BrownJul06_AutomaticallyIdentifyingAndCreatingAcceleratorsDirectlyFromCCode_Xcell.pdf

Cadence Tensilica Processor IP. https://www.cadence.com/en_US/home/tools/ip/tensilica-ip.html

Castrillon J, Zhang D, Kempf T, Vanthournout B, Leupers R, Ascheid G (2009) Task management in MPSoCs: an ASIP approach. In: Proceedings of the 2009 international conference on computer-aided design (ICCAD '09). Association for Computing Machinery, New York, pp 587–594. https://doi.org/10.1145/1687399.1687508

Chattopadhyay A, Kammler D, Witte EM, Schliebusch O, Ishebabi H, Geukes B, Leupers R, Ascheid G, and Meyr H (2006a) Automatic low power optimizations during ADL-driven ASIP design. In VLSI design, automation and test, 2006 international symposium on, pp 1–4,

Chattopadhyay A, Geukes B, Kammler D, Witte EM, Schliebusch O, Ishebabi H, Leupers R, Ascheid G, Meyr H (2006b) Automatic ADL-based operand isolation for embedded processors. In Proceedings of the conference on design, automation and test in Europe: Proceedings, DATE '06, pp 600–605

Chattopadhyay A, Sinha A, Zhang D, Leupers R, Ascheid G, Meyr H (2006c) Integrated verification approach during ADL-driven processor design. In Rapid system prototyping, 2006. Seventeenth IEEE international workshop on, pp 110–118

Chattopadhyay A, Chen X, Ishebabi H, Leupers R, Ascheid G, Meyr H (2008) High-level modelling and exploration of coarse grained re-configurable architectures. In Proceedings of the conference on design, automation and test in Europe, DATE '08, pp 1334–1339

Chattopadhyay A, Leupers R, Meyr H, Ascheid G (2009) Language-driven exploration and implementation of partially re-configurable ASIPs. Springer

Chen X, Minwegen A, Hussain SB, Chattopadhyay A, Ascheid G, Leupers R (2015) Flexible, efficient multimode mimo detection by using reconfigurable asip. IEEE Trans Very Large Scale Integrat (VLSI) Syst 23(10):2173–2186

Chisel Programming Language. https://www.chisel-lang.org

Cmelik B, Keppel D (1994) Shade: a fast instruction-set simulator for execution profiling. In Proceedings of the 1994 ACM SIGMETRICS conference on measurement and modeling of computer systems, SIGMETRICS '94, pp 128–137

Codasip. https://codasip.com

Cox S, Taussig D (2016) Extending RISC-V for application-specific requirements. 5th RISC-V Workshop, November 29, 2016

Dang TN, Roychoudhury A, Mitra T, Mishra P (2009) Generating test programs to cover pipeline interactions. In Proceedings of the 46th annual design automation conference, DAC '09, pp 142–147

Efland G et al (2016) High performance DSP for vision, imaging and neural networks. Hot Chips, Cupertino

Fauth A, Knoll A (1993) Automated generation of DSP program development tools using a machine description formalism. In Acoustics, speech, and signal processing, 1993 IEEE international conference on, vol 1, pp 457–460

Freericks M (1993) The nML machine description formalism. TU Berlin CS Dept. Technical Report TR SM-IMP/DIST/08

Geuzebroek J (2014) Leveraging processor extensibility to build an ultra low-power embedded subsystem. Synopsys white paper

Goodwin D, Pekov D (2003) Automatic generation of application specific processors. Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems (CASES-2003), pp 137–147

Goossens G, Lanneer D, Geurts W, Van Praet J (2006) Design of ASIPs in multi-processor SoCs using the Chess/Checkers retargetable tool suite. Int Symp System-on-Chip. https://doi.org/10.1109/ISSOC.2006.321968

Gries M, Keutzer K (2005) Building ASIPs: the mescal methodology. Springer, Berlin/Heidelberg, Germany

V. Grimblatt, G. Ferré, F. Rivet, C. Jego and N. Vergara, "Precision agriculture for small to medium size farmers – an IoT approach," 2019 IEEE Int Symp Circuits Syst (ISCAS), 2019, pp. 1–5, https://doi.org/10.1109/ISCAS.2019.8702563

Gupta A, Pal A (2015) Accelerating SVM on ultra low power ASIP for high throughput streaming applications. In: 2015 28th international conference on VLSI design. IEEE

Hadjiyiannis G, Hanono S, Devadas S (1997) ISDL: an instruction set description language for retargetability. In Proceedings of the 34th annual design automation conference, DAC '97, pp 299–302

Halambi A, Grun P, Ganesh V, Khare A, Dutt N, Nicolau A. EXPRESSION: a language for architecture exploration through compiler/simulator retargetability. In Design, automation and test in Europe conference and exhibition 1999. Proceedings, pp 485–490, March 1999

Hartoog MR, Rowson JA, Reddy PD, Desai S, Dunlop DD, Harcourt EA, Khullar N (1997) Generation of software tools from processor descriptions for hardware/software codesign. In Proceedings of the 34th annual design automation conference, DAC '97, pp 303–306

Hassan MA, Imai M ASIP Meister: an ASIP design environment. DATE 2005 University Booth, https://www.edacentrum.de/system/files/files/veranstaltungen/2005/date05/ubooth/descriptions/Description_sw_ASIP.pdf

Hohenauer M, Scharwaechter H, Karuri K, Wahlen O, Kogel T, Leupers R, Ascheid G, Meyr H, Braun G, van Someren H (2004) A methodology and tool suite for C compiler generation from ADL processor models. In Proceedings of the conference on design, automation and test in Europe – Volume 2, DATE '04, 2004

Hohenauer M, Schumacher C, Leupers R, Ascheid G, Meyr H, Someren Hv (2006) Retargetable code optimization with SIMD instructions. In CODES+ISSS '06: Proceedings of the 4th international conference on hardware/software codesign and system synthesis. ACM, New York, pp 148–153

Hohenauer M, Engel F, Leupers R, Ascheid G, Meyr H, Bette G, Singh B (2008) Retargetable code optimization for predicated execution. In Proceedings of the conference on design, automation and test in Europe, DATE '08, pp 1492–1497

Husár A, Trmac M, Hranac J, Hruska T, Masarík K (2010) Automatic C compiler generation from architecture description language ISAC. MEMICS:47–53

Inoue A, Tomiyama H, Fajar E, Yasuura NH, Kanbara H (1998) A programming language for processor based embedded systems. In Proc. of APCHDL, pp 89–94

Itoh M, Higaki S, Sato J, Shiomi A, Takeuchi Y, Kitajima A, Imai M (2000a) PEAS-III: an ASIP design environment. In Computer design, 2000. Proceedings. 2000 international conference on, pp 430–436

Itoh M, Takeuchi Y, Imai M, Shiomi A. Synthesizable HDL generation for pipelined processors from a micro-operation description. IEICE Trans. Fundamentals, March 2000b

Jamma D et al (2016) Design exploration of ASIP architectures for the K-nearest neighbor machine-learning algorithm. In: 2016 28th international conference on microelectronics (ICM). IEEE

Jani D, Benson C, Dixit A, Martin G (2005) Chapter 18 Functional verification of configurable embedded processors. In: Bailey B (ed) The functional verification of electronic systems: an overview from various points of view. IEC Press, Chicago, US

Kalluri SS. Securing offload engines for A robust secure SoC system. https://semiengineering.com/securing-offload-engines-for-a-robust-secure-soc-system/

Kammler D, Zhang D, Schwabe P, Scharwaechter H, Langenberg M, Auras D, Ascheid G, Mathar R (2009) Designing an asip for cryptographic pairings over barreto-naehrig curves. In: Clavier C, Gaj K (eds) Cryptographic hardware and embedded systems – CHES 2009. Springer, Berlin, Heidelberg, pp 254–271

Karuri K, Chattopadhyay A, Hohenauer M, Leupers R, Ascheid G, Meyr H (2007) Increasing data-bandwidth to instruction-set extensions through register clustering. In IEEE/ACM international conference on computer-aided design (ICCAD)

Karuri K, Chattopadhyay A, Chen X, Kammler D, Hao L, Leupers R, Meyr H, Ascheid G (2008) A design flow for architecture exploration and implementation of partially reconfigurable processors. IEEE Trans Very Large Scale Integr Syst 16(10):1281–1294

Khare A, Savoiu N, Halambi A, Grun P, Dutt N, Nicolau A (1999) V-SAT: a visual specification and analysis tool for system-on-chip exploration. In EUROMICRO conference, 1999. Proceedings. 25th, vol 1, pp 196–203

Kraemer S, Gao L, Weinstock J, Leupers R, Ascheid G, Meyr H (2007) HySim: a fast simulation framework for embedded software development. In: CODES+ISSS '07: proceedings of the 5th IEEE/ACM international conference on hardware/software codesign and system synthesis, pp 75–80

Lanneer D, Praet J, Kifli A, Schoofs K, Geurts W, Thoen F, Goossens G (1995) CHESS: retargetable code generation for embedded DSP processors. Code Generation for Embedded Processors, pp 85–102

Leupers R, Marwedel P (1998) Retargetable code generation based on structural processor description. Des Autom Embed Syst 3(1):75–108

Leupers R, Karuri K, Kraemer S, Pandey M (2006) A design flow for configurable embedded processors based on optimized instruction set extension synthesis. In: DATE '06: proceedings of the conference on design, automation and test in Europe. European Design and Automation Association, pp 581–586

Leupers R Chattopadhyay A, Dutt N, Mishra P (2016) Processor modeling and design tools. Chapter 9 of EDA for IC system design, verification and testing (Volume 1 of the Electronic Design Automation for Integrated Circuits Handbook, Second Edition), edited by L. Lavagno, I. Markov, G. Martin, and L. Scheffer, CRC Press/Taylor and Francis

Machetti S (2018) ASIP design for motion estimation in video compression algorithms. PhD Thesis. Politecnico di Torino

Makimoto T (2002) The hot decade of field programmable technologies. IEEE international conference on field-programmable technology, 2002. (FPT). Proceedings, pp 3–6. https://doi.org/10.1109/FPT.2002.1188657

Martin G, Nicolaescu D (2018) Enhancing DSP design productivity with automated generators. unpublished paper, 2018. (Available by emailing)

Marwedel P (1979) The MIMOLA design system: detailed description of the software system. 16th Design Automation Conference, pp 59–63. https://doi.org/10.1109/DAC.1979.1600089

Maydan D (2011) Evolving voice and audio requirements for smartphones. Linley Technology Mobile Conference

Meyr H, Chattopadhyay A, Leupers R (2008) LISA: a uniform ADL for embedded processor modelling, implementation and software toolsuite generation. In: Processor description languages, edited by P. Mishra and N. Dutt, pp 95–130

Mishra P (2005) Processor validation: a top-down approach. Potentials, IEEE 24(1):29–33

Mishra P, Dutt N (2004a) Modeling and validation of pipeline specifications. ACM Trans Embed Comput Syst 3(1):114–139

Mishra P, Dutt N (2004b) Graph-based functional test program generation for pipelined processors. In Design, automation and test in Europe conference and exhibition, 2004. Proceedings, vol 1, pp 182–187

Mishra P, Dutt N (2005a) Functional coverage driven test generation for validation of pipelined processors. In Proceedings of the conference on design, automation and test in Europe - Volume 2, DATE '05, pp 678–683

Mishra P, Dutt N (2005b) Architecture description languages for programmable embedded systems. In: IEE Proceedings on computers and digital techniques

Mishra P, Dutt N (eds) (2008) Processor description languages. Morgan Kaufmann Publishers Inc

Mishra P, Dutt N, Tomiyama H (2003) Towards automatic validation of dynamic behavior in pipelined processor specifications. Des Autom Embed Syst 8(2–3):249–265

Mishra P, Kejariwal A, Dutt N (2004) Synthesis-driven exploration of pipelined embedded processors. In VLSI design, 2004. Proceedings of the 17th international conference on, pp 921–926

Nohl A, Braun G, Schliebusch O, Leupers R, Meyr H, Hoffmann A (2002) A universal technique for fast and flexible instruction-set architecture simulation. In Proceedings of the 39th annual design automation conference, DAC '02, pp 22–27

Nohl A, Greive V, Braun G, Hoffman A, Leupers R, Schliebusch O, Meyr H. (2003) Instruction encoding synthesis for architecture exploration using hierarchical processor models. In Design automation conference, 2003. proceedings, pp 262–267

Ogawa HS, Luther TE, Ricardini JE, Cunha H, Simplicio M Jr, Aranha DF, Derwig R, Kupwade-Patil H (2019) Accelerated v2x provisioning with extensible processor platform. Cryptology ePrint Archive, Report 2019/1039, x. https://ia.cr/2019/1039

Pees S, Hoffmann A, Meyr H (2000) Retargetable compiled simulation of embedded processors using a machine description language. ACM Trans Des Autom Electron Syst 5(4):815–834

Podivinsky J, Cekan O, Krcma M, Burget R, Hruska T, Kotasek Z (2018) A processor optimization framework for a selected application. IEEE East-West design & test symposium (EWDTS), pp 1–11. https://doi.org/10.1109/EWDTS.2018.8524733

Pothineni N, Kumar A, Paul K. (2008) Exhaustive enumeration of legal custom instructions for extensible processors. In VLSI design, 2008. VLSID 2008. 21st international conference on, pp 261–266

Press Release. Codasip appoints Ron Black as CEO. 2 December 2021. https://codasip.com/2021/12/02/codasip-appoints-ron-black-as-ceo/

Přikryl Z (2019) Code density improvements beyond the C standard extension. RISC-V Summit

Přikryl Z. Creating domain-specific processors using custom RISC-V ISA instructions. Codasip White Paper, September 23, 2020. https://codasip.com/2020/09/23/creating-domain-specific-processors-using-custom-risc-v-isa-instructions/

Qin W, Malik S (2002) Architecture description languages for retargetable compilation. In: Compiler design handbook: optimizations & machine code generation. CRC Press, Chicago, US, pp 535–564

Rajesh V, Moona R. Processor modeling for hardware software codesign. In VLSI design, 1999. Proceedings. twelfth international conference on, pp 132–137, Jan 1999

Rakossy ZE, Naphade T, Chattopadhyay A (2012) Design and analysis of layered coarse-grained reconfigurable architecture. In Reconfigurable computing and FPGAs (ReConFig), 2012 international conference on, pp 1–6

Rakossy ZE, Aponte AA, Chattopadhyay A (2013) Exploiting architecture description language for diverse ip synthesis in heterogeneous mpsoc. In Reconfigurable computing and FPGAs (ReConFig), 2013 international conference on, pp 1–6

RAO, Jinli et al (2018) Design exploration of SHA-3 ASIP for IoT on a 32-bit RISC-V processor. IEICE Trans Inf Syst 101(11):2698–2705

Reshadi M, Mishra P, Dutt N (2003) Instruction set compiled simulation: a technique for fast and flexible instruction set simulation. In Proceedings of the 40th annual design automation conference, DAC '03, pp 758–763

RISC-V Custom Instructions – Design, Development And Deployment, February 24, 2021. http://www.andestech.com/en/2021/02/24/risc-v-custom-instructions-design-development-and-deployment-%E2%94%82-2021-webinar/

Roeven H, Coninx J, Ade M (2004) CoolFlux DSP-The embedded ultra low power C-programmable DSP core. Proceedings of the International Signal Proceedings of Conference (GSPx)

Rounioja K, Puusaari K (2006) Implementation of an hsdpa receiver with a customized vector processor. In: 2006 international symposium on system-on-chip. IEEE

Rowen C (2012) Power/performance breakthrough for LTE advanced handsets. Linley Mobile Conference, April 16, 2012

Sanghavi H, Andrews N (2008) TIE: an ADL for designing application-specific instruction-set extensions. Processor description languages, edited by P. Mishra and N. Dutt, pp 183–216

Saponara S, Fanucci L, Marsi S, Ramponi G, Kammler D, Witte EM (2007) Application-specific instruction-set processor for retinex-like image and video processing. IEEE Trans Circuits Syst II Express Briefs 54(7):596–600

Schliebusch O, Chattopadhyay A, Leupers R, Ascheid G, Meyr H, Steinert M, Braun G, Nohl A (2004a) RTL processor synthesis for architecture exploration and implementation. In: Design, automation and test in Europe conference and exhibition, 2004. Proceedings, vol 3, pp 156–160

Schliebusch O, Kammler D, Chattopadhyay A, Leupers R, Ascheid G, Meyr H (2004b) Automatic generation of JTAG interface and debug mechanism for ASIPs. In GSPx, 2004. Proceedings

Selim Z, et al An efficient ASIP design methodology. https://www.design-reuse.com/articles/24082/asip-design-methodology.html

Siska C (1998) A processor description language supporting retargetable multi-pipeline dsp program development tools. In Proceedings of the 11th international symposium on system synthesis, ISSS '98, pp 31–36

Synopsys ASIP Designer. https://www.synopsys.com/designware-ip/processor-solutions/asips-tools.html, https://www.synopsys.com/dw/ipdir.php?ds=asip-designer

Synopsys ASIP Designer. https://www.synopsys.com/designware-ip/processor-solutions/asips-tools.html, and https://www.synopsys.com/dw/ipdir.php?ds=asip-designer

Synopsys DesignWare ARC Processor Cores. https://www.synopsys.com/designwareip/processor-solutions.html

The MDES User Manual. http://www.trimaran.org

Tomiyama H, Halambi A, Grun P, Dutt N, Nicolau A (1999) Architecture description languages for systems-on-chip design. In: In the sixth Asia Pacific conference on chip design language, pp 109–116

Trmac M, Husár A, Hranac J, Hruska T, Masarík K (2010) Instructor selector generation from architecture description. MEMICS:109–115

van der Wolf P, Derwig R (2013) Modular SoC integration with subsystems The audio subsystem case. Design, automation & test in Europe conference & exhibition (DATE), 2013, pp 157–162. https://doi.org/10.7873/DATE.2013.045

Wahlen O, Hohenauer M, Leupers R, Meyr H (2003) Instruction scheduler generation for retargetable compilation. Design Test Comp 20(1):34–41. IEEE

Wang Z, Chattopadhyay A (2017) High-level estimation and exploration of reliability for multi-processor system-on-chip. In Springer

Wang S, Malik S. (2003) Synthesizing operating system-based device drivers in embedded systems. In Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis, CODES+ISSS '03, pp 37–44

Wang Z et al (2012) ASIC synthesis using architecture description language. Proceedings of technical program of 2012 VLSI design, automation and test. IEEE

Watson RNM, Moore SW, Sewell P, Neumann PG (2019) An Introduction to CHERI. Technical Report, UCAM-CL-TR-941, University of Cambridge

Witchel E, Rosenblum M (1996) Embra: fast and flexible machine simulation. In: Proceedings of the 1996 ACM SIGMETRICS international conference on measurement and modeling of computer systems, SIGMETRICS '96, pp 68–79

Xie H Wang Z, Wang L, Chattopadhyay A (2013) Power modeling and estimation during adl-driven embedded processor design. In Energy aware computing systems and applications (ICEAC), 2013 4th annual international conference on, pp 97–102

Zhang D, et al (2013) Optimized communication architecture of MPSoCs with a hardware scheduler: a system-level analysis. Adoption and optimization of embedded and real-time communication systems. IGI Global, pp 163–180

Zhu J, Gajski DD (1999) A retargetable, ultra-fast instruction set simulator. In Proceedings of the conference on design, automation and test in Europe, DATE '99

Accelerator Design with High-Level Synthesis

24

Christian Pilato and Stephanie Soldavini

Contents

Introduction . . . 842
Background: Technology and Models . . . 844
Target Technology . . . 844
Accelerator Models . . . 846
Accelerator Template . . . 847
Introduction to High-Level Synthesis . . . 848
A Traditional High-Level Synthesis Framework . . . 848
A Bit of History on Commercial Products and Academic Projects . . . 850
From Input Specification to Intermediate Representation . . . 851
Input Specification and Intermediate Representation . . . 851
Analysis and Optimization of the Intermediate Representation . . . 853
Creation of the Microarchitecture . . . 855
Scheduling and Performance Optimization . . . 855
Binding and Resource Optimization . . . 857
Definition of the Memory Architecture . . . 859
Creation of the FSM Controller . . . 863
RTL Generation and System Integration . . . 863
Code Generation, Evaluation, and Verification . . . 863
System-Level Integration and Optimization . . . 864
Open and Modern Challenges . . . 865
Creation of Domain-Specific Architectures . . . 865
Programmability and System-Level Optimization . . . 867
Hardware Security and Data Protection . . . 868
Conclusion . . . 868
References . . . 869

C. Pilato (✉) · S. Soldavini
Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milano, Italy
e-mail: christian.pilato@polimi.it; stephanie.soldavini@polimi.it

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_19

Abstract

Specialized accelerators can exploit spatial parallelism on both operations and data thanks to a dedicated microarchitecture with a better use of the hardware resources. Designers need to describe such components (including the resources, their interconnections, and the control logic) in proper hardware languages compatible with synthesis tools. This process requires hardware design skills that are uncommon in software programmers. To boost the use of spatial accelerators, software programmers need automated methods, like *high-level synthesis* (HLS), to specify hardware blocks with high-level languages and automatically translate their specifications into the corresponding hardware descriptions ready for synthesis. While HLS is a key enabling technology for the design of complex hardware/software architectures, developing efficient spatial accelerators requires efficient HLS methods to co-optimize performance and hardware cost with a hardware/software co-design approach. In this chapter, we present the current state of the art in high-level synthesis, covering all steps to create the specialized microarchitecture of an accelerator. We also discuss outstanding challenges that can be addressed with the use of HLS.

Keywords

High-level synthesis · System-on-chip · Compiler optimization · Memory architecture · Heterogeneous architecture

Introduction

Technology scaling is coming to an end due to physical limitations on building smaller transistors and keeping all of them active inside the chip (*dark silicon* problem) (Esmaeilzadeh et al. 2012). Designers need novel solutions to increase the performance and reduce the power consumption of computing systems (Horowitz 2014). Modern workloads, such as Big Data and machine-learning applications, exacerbate these issues because they need to process an increasing amount of data to identify hidden relationships and extract valuable knowledge (Pilato et al. 2021). Traditional computing systems are not able to satisfy the performance requirements of these applications, while the large and frequent data transfers lead to unsustainable power consumption. Single, complex, hyper-pipelined processors are thus replaced by parallel architectures with simpler processors (*multicore architectures*) and/or specialized components (*heterogeneous architectures*). This allows designers to significantly reduce power consumption: a specialized accelerator is tailored to execute specific functions, activated only when needed, and can be turned off to save additional static power when unused (Mantovani et al. 2020).

Designing specialized accelerators is complex, expensive, and time-consuming. Designers have to determine the proper microarchitecture to achieve the desired performance with minimal resources. The microarchitecture is usually designed

for *spatial computation*, i.e., to execute operations on multiple data in parallel. The resulting design must be described with a *hardware description language* (HDL) for enabling the actual hardware implementation. However, this process requires specific hardware design skills that are uncommon in application designers. Finally, implementing a specialized accelerator as an application-specific integrated circuit (ASIC) allows designers to achieve the best energy efficiency but limits the reusability of the components and, in turn, the sustainability of the architecture design process. Field-programmable gate array (FPGA) devices are becoming attractive to reduce the cost of accelerator development by reusing the same resources across multiple components after *reconfiguring* the device.

To cope with the increasing complexity of such heterogeneous architectures, designers need to raise the abstraction level for both system and component design. At the system level, custom design flows are replaced by more reusable architectures based on the concept of "platform," like the one shown in Fig. 1. A platform template is progressively refined to obtain the final architecture with proper customizations (Cong et al. 2006). This approach operates at the system level, describing the component functionalities and their interactions above the register-transfer level (RTL), and uses automated methods for component synthesis and integration (Cong et al. 2011). This enables the reuse of the same high-level components across multiple projects, significantly reducing design time and costs (Bombieri et al. 2013). At the component level, manual design is replaced by automated methods based on *high-level synthesis* (HLS), which is the process of automatically translating a behavioral specification into the corresponding RTL description ready for synthesis (De Micheli 1993; Martin and Smith 2009). This process is similar to the generation of machine code with compilers. HLS can be divided into three phases: the analysis of the input description to extract relevant information regarding the functionality to implement, the creation of the optimized microarchitecture that implements the given functionality, and the generation of the final RTL descriptions for the subsequent synthesis steps. This requires the introduction of hardware-oriented concepts (like timing, parallelism, and concurrency) that might not be present in the initial description. In addition,

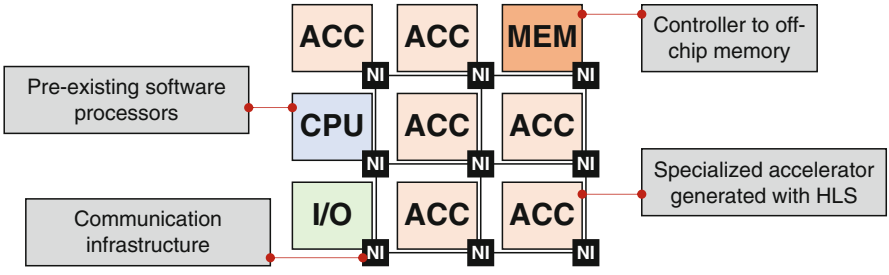


Fig. 1 Example of heterogeneous platform: CPUs and auxiliary elements are predesigned components, while accelerators can be designed with HLS. The FPGA can host the entire system or only some accelerators

the possibility of customizing the target architecture requires a trade-off between performance and use of resources, and the evaluation of the effects of local decisions on the entire design.

This novel boost in *accelerator-rich architectures* is pushed by an increasing number of different computing domains including from datacenters, high-performance computing, reconfigurable embedded systems, and Internet of Things devices (Cong 2015; Cilardo et al. 2015; Windh et al. 2015) with stringent requirements that cannot be met either with processor-based architectures or pre-designed components. So, designers must be able to specify the functionality along with its nonfunctional requirements, derive the final microarchitecture of the components, and create a system that is able to use it with limited changes to the original application. Software and hardware engineers are still speaking different languages with a significant gap even when using HLS (Edwards 2006; Pilato 2017). Although the available HLS tools have a similar organization (Nane et al. 2016), there are several differences in the approaches used for each phase, potentially leading to very different results. Since HLS is mostly application-dependent, it is impossible to determine an optimal flow for every algorithm and computing system. On the contrary, it is crucial to understand algorithms and methods for the different phases to better understand the results and, eventually, how to guide or improve the tools.

This chapter discusses several aspects that lead to create a successful HLS engine, including the specification of the input functionality, the optimization of the microarchitecture, and the system-level integration. It also discusses problems that are still open along with challenges that can be tackled with proper HLS tools. For example, it describes how HLS can enable the creation of more secure components in an era of increasing cyberattacks (Pilato et al. 2018a).

Background: Technology and Models

Target Technology

When selecting the processing elements for an architecture, designers have to face the never-ending challenge of trading off *performance* and *flexibility*, as shown in Fig. 2. While performance is the main optimization goal for many applications, flexibility allows them to reuse the same system for different applications. The traditional **central processing unit** (CPU) is the most flexible component, able to execute any kind of application, sacrificing performance and energy efficiency. A specialized processor, like a **digital signal processor** (DSP), a **graphics processing unit** (GPU), or a **tensor processing unit** (TPU), offers better performance for application-specific workloads (e.g., audio, video, or machine-learning applications) while maintaining a certain degree of flexibility.

A **field-programmable gate array** (FPGA) device is an array of elements that can be configured by the user to execute a specific functionality even after fabrication, as shown in Fig. 3. For this reason, the elements are called *configurable*

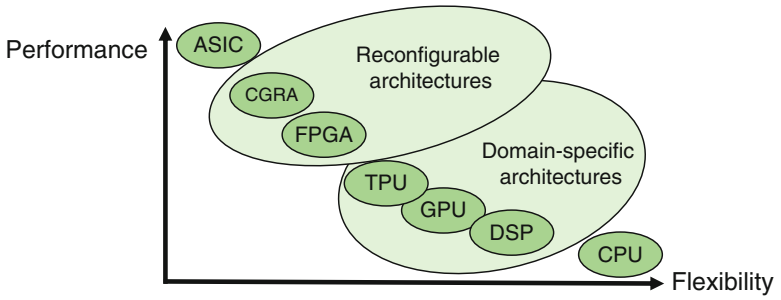


Fig. 2 Trading off performance and flexibility by using different target technologies. (Adapted from Ndu 2012)

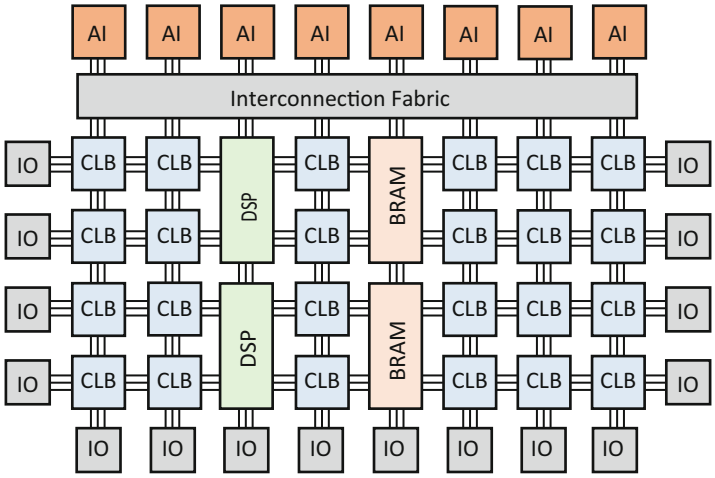


Fig. 3 High-level FPGA organization: the device contains an array of configurable elements and heterogeneous resources (e.g., DSP and Block RAMs). It may also feature dedicated engines that are designed and optimized for AI processing

logic blocks and their configuration is called a *bitstream*. FPGAs have the flexibility of processor-like architectures since they can be reused (after reconfiguration) to execute different workloads, but they can achieve performance comparable to ASICs, thanks to the possibility of implementing specialized microarchitectures on the configurable blocks. Some FPGA devices also offer the possibility of changing their functionality *during* the execution through *partial dynamic reconfiguration*. Designers create all *partial bitstreams* statically (i.e., partial configurations only for the specific FPGA region where the accelerators will be placed). The reconfiguration loads the proper partial bitstream into the corresponding region through a specific port, called an *Internal Configuration Access Port* (ICAP). This process is time-consuming and used only when the benefits of hardware accelerators are much greater than the reconfiguration time. A **coarse-grain reconfigurable**

array (CGRA) is a network of specialized data processing units. Executing an application requires to create only a configuration of the interconnections. It can achieve better performance than FPGA devices but it is more application specific. In an **application-specific integrated circuit** (ASIC), the microarchitecture of the component is tailored to execute only the given functionality. Specialization allows the circuit to achieve the best performance but limits the reuse and therefore the flexibility of the component. Also, since the functionality of an ASIC cannot be changed after fabrication, manually tuning and verifying the design leads to high design costs. In all cases, it is important to understand how the operations are implemented in hardware to better understand the use of resources. For example, ASIC designs are mapped onto standard cells or hard macro blocks (like memories), and the silicon area is a good metric to characterize the implementation. On the contrary, FPGA designs can use a heterogeneous set of resources, like configurable logic cells, Block RAMs, and DSP blocks. In this case, comparing two designs is much harder.

Accelerator Models

Designers have several alternatives for creating the microarchitecture of the specialized accelerators, especially due to different execution modes (Cota et al. 2015). *Configurable, extensible processors* have specific accelerators integrated into the pipeline of the given CPU to improve the execution of specific code portions. The selected code to be accelerated is represented with a new instruction, and, for this reason, the components are commonly referred to as *custom instruction set extensions* (Brisk et al. 2004). After selecting the kernel instructions and extending the compiler to target them (Galuzzi et al. 2006), HLS can create the RTL microarchitecture of the accelerator to be integrated in the processor datapath (Pothineni et al. 2010). Commercial products like **Synopsys ARC** and **Cadence Xtensa** feature complete toolchains to profile the application, identify hot-spots to be accelerated, and integrate the corresponding RTL modules. *Accelerator-rich architectures* feature, instead, several stand-alone components that are designed to provide peak performance for selected large kernels or even complete applications. Large data sets are usually stored in an external memory (e.g., DRAM) that is accessed with specific interfaces. Such accelerators can be *configurable* with parameters that the user can specify (usually through input ports) to select a specific functionality of the component. While components can be extremely complex, many modern applications can be reduced to operations on large streams of data. This is, for example, the case of machine-learning applications that need to perform simple operations (e.g., convolutions) on many data. Control constructs and synchronization can be introduced to reuse the same hardware resources to iteratively operate on new data using the same concepts as software loops. To reduce the complexity of large designs, designers can create *dataflow architectures* as a collection of components communicating with latency-insensitive protocols (Carloni 2015). The specialization of the components can target the composition of the block (Edwards et al. 2019), the memory architecture (Pilato et al. 2017), or

the communication primitives (Guglielmo et al. 2014). Modern machine-learning applications benefit from architectures with simple accelerators that are fed with streams of data and reused across multiple cases, like *modular accelerators* for neural networks (Venkatesan et al. 2019) and *dataflow systolic arrays* for matrix multiplications (Liu and Cong 2019). For these applications, modern FPGAs are implementing specific vector processor cores, like the AI engines in the Xilinx Versal AI Core Series (Chatarasi et al. 2020).

Accelerator Template

Each specialized accelerator is composed of two fundamental blocks: a *datapath* and a *controller* (Zhu and Gajski 1999). The **controller** can be modeled as a finite-state machine (FSM) that determines which operations execute in each clock cycle and sends control signals to trigger the operators and the interconnection in the **datapath** to perform the computation. The datapath is a collection of interconnected hardware resources that can execute in parallel (spatial computation). Additional components include local and external memories for data *storage*. Internal memories are accessed directly by the datapath resources, while external ones are accessed through pre-defined components like memory controllers. When the complexity increases, modules are organized in submodules with the same structure, like in software functions. Usually, common functions are replicated at different levels of the hierarchy, like uniquification in logic synthesis, although solutions attempt to reuse the same hardware blocks with special proxies to eliminate those function copies and reduce resource requirements (Minutoli et al. 2015). Accelerators are usually encapsulated in an infrastructure to interact with the system, as shown in Fig. 4. They can be also interconnected with each other for direct communication. This architecture is often used for dataflow applications, where data are streamed

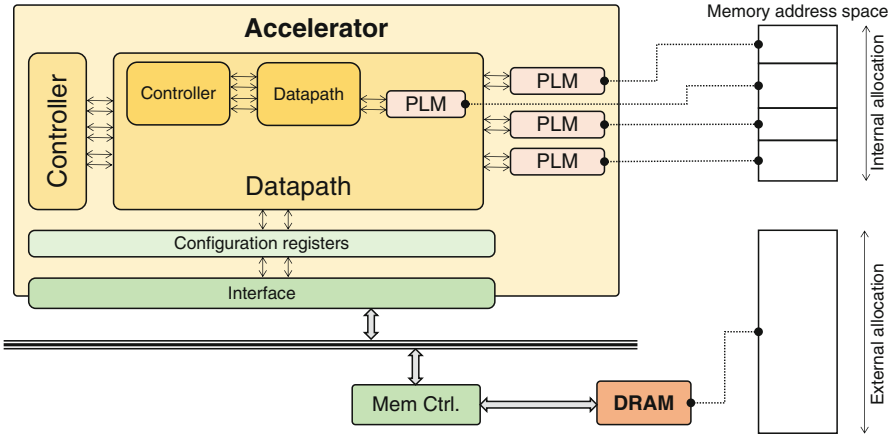


Fig. 4 An accelerator is composed of submodules hierarchically organized, and may contain *private local memories* to store local data and be connected to the external memory

from one component to the next (Giri et al. 2020). The processor core (CPU) executes a software application to prepare the data in memory and configure the accelerator through the interconnection system (e.g., a bus or a network-on-chip) with memory-mapped operations on the **configuration registers** that are connected to the input ports (Mantovani et al. 2020). The data stored in the external memory (DRAM) are accessed through one or more memory controllers (Mantovani et al. 2016a). DMA mechanisms allow accelerators to exchange large data blocks with DRAM (Cota et al. 2015; Pilato et al. 2017). When moved inside accelerators, the data are stored in **private local memories** (PLMs) for fast access. PLMs can also store data for the entire execution of the accelerator (Pilato et al. 2011b). Accelerators access PLM data with known latency (e.g., one or two cycles), while the latency of external accesses is usually unpredictable. While PLM accesses make the scheduling of memory operations and the controller creation simpler, accelerators must implement a latency-insensitive memory interfaces to guarantee execution correctness when accessing external data. An FPGA can host the entire system or only the accelerator part. In the latter case, the FPGA is combined with a hard-core processor through an interconnection fabric or is a stand-alone component, like in the IBM *cloudFPGA* project (Weerasinghe et al. 2016). IP and technology vendors provide intellectual property (IP) blocks for common functions. For example, Synopsys and Cadence offer soft IPs for high-speed communication (e.g., SerDes IPs) like Ethernet physical layers. Similarly, FPGA vendors offer a list of configurable IPs for common peripherals like DMA controllers to exchange data with external memory banks, to access USB ports and PCIe bridges, and to display data through video controllers.

Introduction to High-Level Synthesis

The design of specialized hardware accelerators requires a design flow that allows designers to generate the register-transfer level (RTL) microarchitecture associated with the desired functionality. High-level synthesis is a key technology in this context. It automatically translates an input high-level specification into the corresponding RTL implementation ready for logic synthesis (either for ASIC or FPGA technologies). **High-level synthesis** is, indeed, a collection of methods and algorithms to automatically define the RTL microarchitecture of a hardware module starting from the specification of its functionality at a higher level of abstraction. This process is similar to the generation of machine code for programmable processors by compilers.

A Traditional High-Level Synthesis Framework

Figure 5 shows the overall organization of a classic HLS flow. The *high-level code* represents the input functionality to implement. The designer can start from an existing algorithm described in software-like languages (e.g., C, C++, or Python),

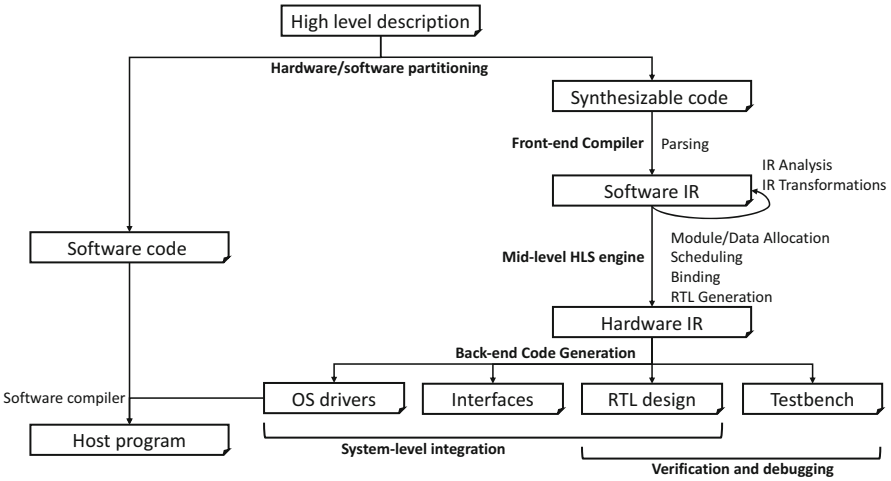


Fig. 5 Overall organization of a classic HLS flow

hardware-oriented languages (e.g., SystemC), or domain-specific languages (e.g., Chisel). Each language targets specific application domains or designer cases (see Section "Input Specification and Intermediate Representation" for more details). First, the input code is split into two parts (**hardware/software partitioning**): the code to be executed by a software processor and the part to be implemented in hardware. The former follows a classic compilation flow to create the binary executables, while the latter must be translated into the corresponding hardware description and integrated with the CPU.

This input description is processed by a front end to remove language-dependent details and extract the *essential semantics*, which is represented through a more generic *intermediate representation* (*SW IR*). Since the IR impacts the following steps, the front-end phase includes several **IR transformations** to obtain a more hardware-oriented representation.

In the mid-level phase, the core of the HLS engine creates the accelerator microarchitecture, i.e., the list of hardware submodules and their interconnections. The HLS engine is composed of several steps. **Scheduling** defines *when* each operation can start its execution to satisfy dependencies and maximize hardware parallelism. **Allocation** and **binding**, instead, determine *where* and *how* each operation is executed. For example, module allocation determines the number of functional units, while module binding assigns each operation to the proper resource to avoid conflicts. The same applies to data and memories. **Data allocation** includes the definition and specialization of the memory architecture to efficiently store and move the data to reduce bottlenecks, i.e., the situations where computation resources are stalling because data are not available. Finally, **RTL generation** creates a representation of the resulting microarchitecture (*HW IR*), along with the logic to control the execution.

Finally, the back-end phase produces the artifacts for the subsequent design steps. First, **code generation** produces the target RTL description in the desired hardware description language (HDL) (*RTL design*). Similarly, **testbench generation** and **interface generation** produce elements to support system-level integration and verification of both the component and the system, respectively. This entire process can be part of a larger design-space exploration framework to trade-off the different design objectives within the final system.

A Bit of History on Commercial Products and Academic Projects

The first HLS projects targeted ASIC with so-called silicon compilers (Gajski 1984; Brewer and Gajski 1990), used especially for simple, data-intensive applications. For example, **Chippe** (Brewer and Gajski 1990) included layout constraints during behavioral synthesis. As the complexity of the hardware modules started increasing, designers shifted toward high-level languages and compiler-based HLS frameworks (Bazargan et al. 2000). Also, since FPGA devices allow for a fast turnaround time to achieve a solution while ASIC requires extensive fine tuning of the designs, HLS tools have been mostly developed for FPGA (Cong et al. 2011), with several academic prototypes and commercial solutions (Nane et al. 2016) (see ▶ Chap. 28, "FPGA-Specific Compilers" for more details on FPGA HLS tools).

Commercial tools are more oriented to a horizontal approach, simplifying coding, porting, and analysis of the solutions. In most of the cases, such tools are offered by FPGA vendors to simplify the use of their devices, in some cases free of charge. For example, **Xilinx Vivado HLS** targets Xilinx FPGA devices, and **Intel HLS Compiler** produces RTL code for Intel FPGA devices. Other tools, like **Siemens EDA Catapult HLS** or **Microsemi LegUp HLS Compiler**, are not vendor specific and can target a broader range of devices. Some HLS tools also target ASIC, like **Cadence Stratus**, **Siemens EDA Catapult HLS**, and **NEC CyberWorkBench**. All these tools have graphical user interfaces or TCL scripts to automate the steps, with good estimators for performance and resource usage. Indeed, they are often tightly connected to logic synthesis tools to provide accurate resource characterizations, especially in case of ASIC. Most HLS tools also provide synthesizable libraries of communication and synchronization protocols for building more complex systems by focusing only on computational aspects (Guglielmo et al. 2014).

Academic projects, instead, are usually more focused on research and experimentation with a vertical approach. For example, **Spark** (Gupta et al. 2003) was the first *public* HLS framework, where the designer could set constraints on the resources to show the relevance and impact of compiler transformations. **GAUT** (Coussy et al. 2008) was a framework for DSP applications. GAUT introduced the concepts of memory mapping, communication modules, and I/O timing to create pipelined architectures. **xPilot** (Cong et al. 2006) was the first project to provide a complete framework for the synthesis of application-specific configurable processors and heterogeneous multicore architectures, focusing on FPGA targets. xPilot had been

later acquired by Xilinx to become the base of Vivado HLS, becoming one of the most complete and easy-to-use HLS tools on the market. **LegUp** (Canis et al. 2013) was an open-source HLS framework based on LLVM that allowed the creation of complete SoC architectures for Altera (now Intel) FPGA devices. Thanks to its modular organization, it has been widely used to prototype different types of solutions for HLS problems, like bit-width analysis (Klimovic and Anderson 2013), profiling-driven optimization (Hadjis et al. 2015), and the effects of compiler optimizations (Huang et al. 2015). It is now discontinued as it became a commercial product, Microsemi LegUp HLS Compiler. **Bambu** (Pilato and Ferrandi 2013) is one of the remaining open-source HLS frameworks. It allows designers to experiment with HLS solutions, thanks to a modular and dynamic compilation framework based on both GCC and LLVM (Lattuada and Ferrandi 2019). It focuses on the problem of understanding how to synthesize C/C++ semantics. To do so, it offers a unique memory microarchitecture that supports most of the C constructs without semantic changes (including dynamic pointer resolution and memory allocation (Pilato et al. 2011b)). It has been also used to integrate solutions for hardware- and hardware-assisted security, like intrinsic dynamic information flow tracking (Pilato et al. 2019), IP watermarking (Pilato et al. 2018b), and algorithm-level obfuscation (Pilato et al. 2018c). There are also projects that focus on specific accelerator models (like dataflow compilers) or application domains (like deep learning), especially for FPGA targets. The interested reader can refer to the ▸ Chap. 28, "FPGA-Specific Compilers" for more details.

From Input Specification to Intermediate Representation

This section discusses the transition from the input high-level specification into a language-agnostic representation that is optimized for hardware generation. One of the first major challenges in HLS is the right choice of input language and compiler (with associated transformations) based on the characteristics and requirements of the given application (e.g., the expected latency).

Input Specification and Intermediate Representation

Most preexisting algorithms are described with traditional languages (like C, C++, Fortran, etc.). Therefore, modern HLS tools are mostly built on top of state-of-the-art software compilers, like **LLVM** and **GCC** (Buyukkurt et al. 2011; Huang et al. 2015). These compilers have support for many input languages and can support the porting of legacy code into hardware. Also, they can leverage many years of research in compiler construction to extract, analyze, and optimize a representation that is more suitable to hardware implementation. To simplify compiler optimizations, the IR is usually translated into a *static single assignment* (SSA) form where multiple assignments to the same variable create different versions, as shown in Fig. 6. Source-to-source compilers can rewrite existing code into a more hardware-friendly

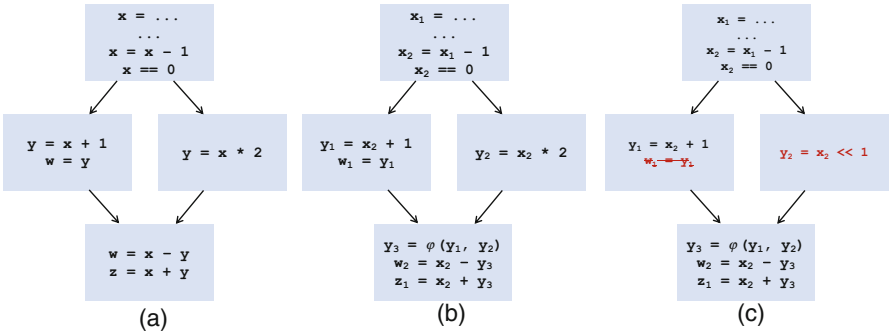


Fig. 6 Example of code (**a**), its corresponding static single assignment (SSA) form (**b**), and the optimized IR after strength reduction and dead-code elimination (**c**)

format and expose more "knobs" for optimization (Cong et al. 2016). For example, since commercial HLS tools use directives to optimize the input code, source-to-source transformations can automatically insert identifiers (e.g., loop labels) and transform the code to better apply synthesis directives.

Modern machine-learning applications make heavy uses of operations on multi-dimensional arrays, also called "tensors." These applications are often extremely parallel and suitable for hardware acceleration. In such systems, the creation of the memory architecture demands efficient methods to describe the operations and the data access patterns. Many languages have been proposed as the front end to HLS frameworks to better expose such details. **Halide** (Ragan-Kelley et al. 2013) simplifies the descriptions of high-performance image and array processing code, while **Halide-HLS** (Pu et al. 2017) is an extension to target FPGAs. Machine-learning applications are built almost exclusively with Python-based frameworks like **PyTorch**, **TensorFlow**, **Caffe**, etc. Python is a popular language that hides many details from the programmers. Compilers translate Python representations into code that can be processed by HLS tools based on traditional IRs.

Traditional compilers progressively transform the IR into simpler operations that are later mapped on machine instructions. This process loses important information for hardware generation, like information of the size of the data structures and the patterns across operations that help define the memory system. Designers are working to extend these representations to pass more information over the compiler passes until it reaches the HLS flow. For example, Google recently proposed LLVM **Multi-Level Intermediate Representation** (MLIR) (Lattner et al. 2020) to create a customizable compilation framework that provides information at different levels. Similarly, **Heterogeneous Parallel Virtual Machine** (HPVM) (Kotsifakou et al. 2018) is a representation for a parallel compiler that aims at simplifying the code implementation of parallel hardware. These representations are often combined with domain-specific languages where the designer can abstract specific hardware details. For example, **Spatial** (Koeplinger et al. 2018) is a recent language to describe hardware accelerators at a higher level. Such descriptions are later compiled and

translated into **Chisel** and then into Verilog. These frameworks can be considered more as "hardware generators" rather than complete HLS tools. Indeed, they operate more as "translators" from the input to the output descriptions, with limited optimizations.

Analysis and Optimization of the Intermediate Representation

The next phase analyzes and transforms the IR extracted from the source code to create a more hardware-friendly representation and optimize the component to generate. Applying IR-level transformations simplifies the following HLS steps, improving the accelerator's performance or reducing its hardware cost. Some transformations are borrowed from traditional compiler optimizations, while others are specific for hardware. This is another motivation for which it is convenient to base HLS on stable and mature compiler frameworks.

Constant propagation and **strength reduction** are classic compiler transformations that can simplify or even eliminate arithmetic operations in the code. For example, the instruction "`x_2 * 2`" of Fig. 6 can be transformed into"x_2 « 1" (see Fig. 6c): a left shifter is much more hardware efficient than a multiplier. Designers may replace some variables with constants representing their average values to leverage these optimizations. These transformations are usually referred as **software approximation techniques** and allow designers to obtain efficient hardware despite minor errors in the results. **Dead-code elimination** removes unnecessary code, which will be otherwise translated into unnecessary hardware (see, e.g., instruction "`w_1 = y_1`" in Fig. 6c). This applies, for example, when control code depends on input parameters that, in specific accelerator instances, are always set to constant values. HLS is often limited by control constructs, like `if-then-else` statements. Operations in the `true`/`false` branches cannot start their execution until the condition is evaluated. **Code speculation** moves some operations before the condition evaluation so that they can be executed in parallel (Lattuada and Ferrandi 2015). Results are temporarily stored in registers and, after evaluating the condition, the values of the "wrong" operations are discarded. This optimization increases the available parallelism, leading to better performance.

Compiler analyses and transformations can also reduce data dependencies and increase hardware parallelism. For example, pointers are widely used to create efficient software code. However, their implementation in hardware is complex because a pointer-based operation must be connected to all the memory locations (either internal or external) where the corresponding information is potentially stored. **Alias analysis** helps determine if two pointers in the source code can ever refer to the same memory location. If it can be proven that two pointers never refer to the same object, there is no dependency, enabling more memory optimizations. **Static pointer resolution** determines the exact variable accessed by a pointer operation, eliminating the need for an explicit pointer. This information can be later used

to optimize the creation of the memory architecture (see Section "Creation of the Microarchitecture") because the two operations can potentially run in parallel when proved to access different data (Pilato et al. 2011b). Additional **memory analyses** determine the list of data structures to allocate in memory and their characteristics (e.g., size and bit-width) for determining whether the corresponding memories fit inside the area constraints of the accelerators. Other transformations operate on the data structures to expose more parallelism. Indeed, arrays are generally stored in memories with limited ports. Designers can apply **array partitioning** and **scalar replacement of aggregates** to reduce the number of memory dependencies.

While software developers can overestimate the bit requirements for some data, many applications do not use the full range of the corresponding variables. Since a processor's hardware is already built with pre-defined registers and arithmetic-logic units, the execution with overestimated variables has almost no extra cost. Hardware specialization requires, instead, to determine the minimal resources needed for the computation to trim unnecessary logic. Therefore, the front-end phase also performs **bit-width analysis** to determine the required precision of each operation to maintain execution correctness and **bit-width transformations** to propagate the information through the design with iterative methods. Similar transformations include also **numerical conversions** (e.g., from floating-point to fixed-point representations) to reduce hardware cost but maintain a certain level of accuracy of the results. In this context, many HLS tools, for example, *Xilinx Vivado HLS* and *Cadence Stratus*, allow library extensions to manually specify the precision of input/output data or to specify particular bit-level operations on the signals. This feature is particularly useful when HLS is used to generate components to be integrated in larger specialized systems like industrial machines. Synthesizable C++ libraries, like **HLSLibs**, have been proposed to extend existing HLS tools with custom precision.

Especially in data-intensive applications, loops account for most of the accelerator execution, and it is complex to extract parallelism from their representations. Most of the parallelism is often between consecutive iterations. While spatial execution can create multiple parallel instances of the loop body, hardware synthesis is usually limited by the loop boundaries. Therefore, loop transformations are widely used to expose more hardware parallelism. For example, **loop unrolling** replicates multiple instances of a loop body to execute in the same iteration. Therefore, the number of operations between control branches is increased, potentially leading to more parallelism. However, this transformation requires a careful analysis of the dependencies; otherwise, the multiple iterations in the same loop body are serialized without an effective speedup. *Artificial dependencies* can also be due to conflict on resources, like in the case of limited memory ports, forcing the serialization of the operations. Another important transformation is **loop pipelining**: consecutive loop iterations are partially overlapped. This optimization follows the same principles of instruction execution in pipelined processors, when an instruction can start before the termination of the previous one. In this case, the *initiation interval* (II) is an important parameter: it represents the number of cycles required by the loop to start a new iteration. A perfect pipeline starts a new iteration after each cycle ($II = 1$).

In case of large data sets, **loop vectorization** aims at executing operations on multiple data in parallel, increasing the demand of memory bandwidth. Other transformations like **loop fusion** and **loop switching** optimize consecutive loops. The interested reader can refer to de Fine Licht et al. (2021) for more details on these loop-related HLS transformations.

Most of the information extracted in this phase is passed to the next steps as extra annotations or directly embedded into the IR. These transformations aim at generating optimized hardware but are often strictly interdependent. For example, loop transformations are not able to expose much parallelism if not properly supported by memory optimizations. Similarly, constant propagation can enable further dead-code elimination. Some HLS tools, for example, *Bambu*, implement *dynamic compilation flows* that re-execute passes whose outcome invalidates previous results or activates further optimizations (Lattuada and Ferrandi 2019).

Creation of the Microarchitecture

To create the microarchitecture of the accelerators, the HLS middle-end requires both the temporal and spatial distribution of the operations to be determined. In the former case, HLS determines *when* to execute each operations, i.e., in which clock cycle, to satisfy any dependency. In the latter case, HLS determines *where* to execute the operations, i.e., on which hardware resources, to minimize the hardware cost while avoiding any conflict.

Scheduling and Performance Optimization

After defining and optimizing the intermediate representation of the functionality to synthesize, the first step is to determine a set of available resources and introduce the concept of *timing*. This process is highly dependent on the previous compiler phase, so they are often executed in an iterative way until the designer reaches a good trade-off between latency and resource usage. **Allocation** determines how many resources will be used for the given component. Since operations executed in the same clock cycle require different resources for the execution, allocation can limit the number of operations that can execute in parallel, i.e., in each clock cycle. **Scheduling** assigns operations to the clock cycles to balance performance and resource usage. Also, scheduling must take into account technology-dependent information like the latency of the hardware modules where the operations are assigned for execution. Temporal assignment must also respect several types of dependencies among operations. HLS dependencies include *real dependencies*, like data dependencies, but also *artificial dependencies* that do not carry real values but are needed for the correct execution of the specification. The result of an operation must be served to the following operation (if feasible in the given clock period) or stored in a register for use in the next clock cycle. This requires the latency of the circuit to be analyzed and compared with the **clock period** (the available timing

budget for each synchronous event), i.e., by estimating the **slack** of each clock cycle (Chang et al. 1996). The slack is the margin between the delay of the circuit and the given timing requirement. Negative slack means that the timing constraint is violated, while a positive slack means that an extra delay could be tolerated. There are multiple situations during scheduling:

- the operation terminates much earlier than the clock period (i.e., the *slack* is positive and high); in this case, it might be possible to execute another operation provided that it fits in the remaining time (**operation chaining**);
- the operation terminates right before the end of the clock period; the result will be then used in a subsequent cycle and it must be stored in a register;
- the operation takes more than one cycle and its result shall be saved only when finished (**multi-cycling**); in this case, if the functional unit is pipelined (i.e., it contains internal register to create computational stages), another operation can start on the same resource even before the current one is completed.

Scheduling is an NP-complete problem and considers different aspects to generate a *valid* implementation (i.e., a circuit that does not produce computational errors). It is usually applied at the basic block level to extract more parallelism. A *basic block* is a sequence of instructions with a single entry-point and no internal branches. The basic block definition is induced by the control constructs contained in the code. Each operation must start its execution only after its predecessors have produced the corresponding results. Then, there must be a physical resource that is able to implement the given operation and is not already in use. So, common HLS tools use heuristic methods to obtain efficient solutions in a reasonable amount of time. Common scheduling algorithms include **list-based scheduling** (Stok 1994), which maintains a list of "ready" operations and progressively assigns them to the clock cycles, and **system of difference constraints (SDC) scheduling** (Cong and Zhang 2006; Canis et al. 2014), which operates on a rich set of scheduling constraints. List-based scheduling is simpler, faster, and more efficient on pure datapath descriptions, while SDC scheduling achieves better results for loop descriptions (Lattuada and Ferrandi 2015).

Exact methods are still applied in case of control-based designs, where the scheduling problem is combined with code motion to execute part of the function speculatively even before a control condition is evaluated (code speculation) (Dai et al. 2018). An alternative approach includes the use of exploration algorithms (e.g., genetic algorithms and particle swarm optimization) to evaluate a variety of solutions. These methods are more efficient since they can find a combination that is more specific for the application but are time-consuming (Pilato et al. 2008).

Executing memory operations in the same clock cycle requires that such operations are independent and there are no conflicts when accessing the data. Conflicts can be avoided by accessing different memory resources or by having memory resources with multiple ports. However, the latency of memory operations depends on where the data are allocated. Accesses to local data have fixed latency (one or two cycles depending on the latency of the memory), and the corresponding memory operations can be considered as other operations. When the data are

allocated outside the accelerators, the HLS engine must follow safe assumptions to guarantee correct execution in all cases: the corresponding data could be allocated off-chip or multiple accelerators can access the same memory creating contention and additional latency. Therefore, the scheduling algorithm must assume the external memory operations have unknown latency (Ranjan Panda et al. 1998). The same case applies to operations corresponding to unpredictable components, like data-dependent submodules. In case of operations with unknown latency, the scheduling uses **latency-insensitive protocols** (Carloni 2015) to guarantee that the computation proceeds only when the operation is completed. However, executing multiple operations with variable latency in the same clock cycle is complex. Therefore, these operations are generally serialized by most of the HLS engines. This serialization may become inefficient when trying to optimize the latency or guarantee the worst-case execution time. Designers proposed approaches for dynamically scheduling the operations. While this concept is highly efficient, the area overhead for implementing the control logic is high and the dynamic scheduling is usually limited to specific cases, like memory-related operations (Pilato et al. 2011a; Josipović et al. 2018).

Binding and Resource Optimization

After introducing timing inside a functional specification, it is necessary to determine which physical resources are effectively used in the target microarchitecture. The following **binding** phases aim at reducing the amount of hardware modules by defining the possibilities of *resource sharing* (Ku and Micheli 1991). This process includes also the definition of the memory architecture, including the partitioning of the data. However, this aspect is discussed in Section "Definition of the Memory Architecture". The binding phase includes the following steps:

- *functional-unit binding* defines which functional unit is used to execute each operation of the specification. It must ensure that each operation is assigned to a functional unit without conflicts, i.e., different operations are not executed by the same functional unit in the same clock cycle.
- *register allocation and binding* determine which data values cross the cycle boundaries and must be stored locally into temporary registers. It also determines how many physical registers must be effectively used (exploiting reuse whenever possible) and how the data values are assigned to them (Brisk et al. 2006).
- *interconnection binding* determines how to connect the datapath resources, which resources are needed to multiplex the signals in case of different paths coming to the same input port of a resource (either functional unit or register), and the corresponding control signals to generate in each clock cycle to correctly route the signals based on the operations to execute.

Several aspects may impact the final implementation, demanding specific approaches. For example, in case of process variation, scheduling and binding

must be considered together with statistical approaches. Execution paths in the specification may have a different probability of execution, and most-executed paths could be more optimized in terms of latency and resource usage. Operations executing in the same clock cycle require distinct functional units to avoid conflicts. Conversely, operations executing in different clock cycles are *compatible*, and, if they are of the same type, they can share the same functional units to reduce resource usage. The *resource binding* problem requires the definition of this set of compatibilities.

Register binding is similar to the process of assigning temporary values to processor registers. The main difference is that, in HLS, there is the possibility of customizing the microarchitecture to add more registers when needed. So, register spilling is not necessary. However, it is still necessary to compute the *liveness* of each variable to determine when different values are compatible and can share the same register. Liveness analysis determines in which points each variable contains a valid value that must be stored in a register. The liveness of a variable v is defined as the interval of time (and, thus, the corresponding clock cycles) between the definition of v and its final use. This interval defines the time for which the value must be stored in a register. Given this definition, two variables u and v are *compatible* if their liveness times are not overlapping. Compatible variables can be stored in the same register because there will be never a moment when both values are needed. Liveness analysis is thus a critical step in register binding. In SSA-based representations, each version can be considered a new variable, and the liveness intervals of these new variables have no interruptions. The register binding can be defined more easily and efficiently. Register binding must also consider the technology for implementing the registers and their impact on the final power consumption of the circuit. For this reason, architectures with multiple supply voltages have been proposed and register binding has been adapted for these cases. Special registers, like razor flip-flops (Ernst et al. 2003), are used to tolerate variations in the latency of the operations to avoid timing violations.

Both these problems are usually represented with a **compatibility graph**. Two operations or two values are compatible (i.e., they are connected with a compatibility edge) when the following two conditions are verified: (1) the two operations can be executed by the same functional unit or the two variables can be stored in the same register, and (2) there are no timing conflicts. A compatibility graph is dual to a **conflict graph**, and the approach to generate them must be conservative to guarantee correctness in every circumstance. Analyses and transformations can remove an edge from a conflict graph or add an edge to a compatibility graph when the property holds in every case. A compatibility problem described with a compatibility graph can be easily solved with *clique covering* formulations, while the dual conflict problem is solved with *coloring* formulations.

This phase must also consider the effects of scheduling on all combined resource binding problems. For example, executing multiple operations in parallel improves the execution time but usually requires more functional units (since operations cannot share the same resources) and registers (since multiple values are produced in the same clock cycle). In addition, resource sharing usually creates multiple paths to the input ports of both functional units (since different operations may require values

from different sources) and registers (since different variables could be produced by distinct functional units). When a port receives signals from multiple sources, HLS engines introduce multiplexers to determine the path active at any given time and drive the signal values. The controller FSM generates control signals to activate the proper paths from source to destination in each clock cycle (see Section "Creation of the FSM Controller"). Resource binding has a huge impact on the number of multiplexers to add. Several methods have been proposed to consider the impact of interconnections during HLS. For example, register binding can be combined with **port swapping** (Chen and Cong 2004). This optimization swaps the inputs of commutative operations, aiming at reducing the number of paths to each port of the units. This reduces, in turn, the number of multiplexers that are needed.

Definition of the Memory Architecture

Nowadays, memory optimization is one of the most important aspects of accelerator design since these components need to process a huge amount of data. However, the accelerators can store on chip only a limited amount of data, usually orders of magnitude less than the total amount. When accelerators operate on more data than can be stored on-chip, they also need to interface with an external memory (e.g., DRAM). This must be taken into account when defining the microarchitecture to avoid executing operations when the data are not available yet but also to hide the communication latency. Latency-insensitive protocols in the interfaces guarantee correct execution in case of unpredictable latency. *Multiple memory channels* and *ping-pong buffers* can hide the latency in accessing the data by parallelizing the data transfers and partially overlapping computation and communication. In case of predictable computation, *pre-fetchers* can anticipate data transfers. These solutions are especially applied in specific application domains, like DNN accelerators, where the structure of the layers can provide information to optimize the architectures. For example, having information on the layers allows optimizations to share buffers and reduce resource utilization with liveness and time span analysis.

The huge latency from external memory access can be mitigated using specialized memory architectures inside the accelerators, such as **caches or private local memories**, as shown in Fig. 7. These components, however, must be co-designed

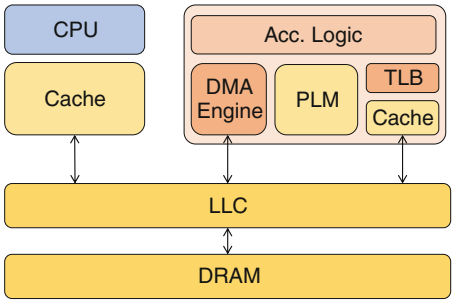


Fig. 7 Memory architecture for specialized accelerators: it can feature caches (with the same principles as CPUs) and private local memories (for fast and deterministic accesses). The elements can share a last level of cache for better performance

accordingly with the algorithms, and optimizing the corresponding architectures requires an in-depth analysis of the memory behavior of the application. For example, solutions have been proposed to include specialized caches where each array of the input specification that is mapped to external memory has its own cache. But the designers need to apply them only to accesses that can guarantee a certain degree of temporal and spatial locality. In this way, the accesses to different arrays are executed in parallel, and the caches mitigate the memory access latency as the memory hierarchy in traditional CPUs. A private local memory is a memory that resides on chip and offers fixed-latency data accesses since it has no logic to support misses. Critical or frequently accessed data can be placed in private local memories to ensure low latency access. The memory behavior of an algorithm is application-dependent, ranging from statically predictable patterns (e.g., stencils for multimedia applications) to irregular memory accesses (e.g., pointer chasing for graph analytics). Compiler transformations based on **polyhedral models** can optimize the memory accesses in case of predicable patterns. Representing operations with polyhedral models allows designers to apply *affine transformations* to make the iterations independent and extract more hardware parallelism. However, to provide enough memory bandwidth, these transformations must be combined with proper **multi-port memory architectures**. Memory IP blocks, like Block RAMs for FPGA or Static RAMs for ASIC, have a limited number of ports. Such architectures provide the requested data in a fixed amount of cycles when the accesses are distributed to different ports, i.e., there are no conflicts. By partitioning memory into separate physical memories, more data can be accessed simultaneously as more ports are available (*memory banking* – see ▶ Chap. 28, "FPGA-Specific Compilers" for more details). The designer must decide how to partition the data structures at design time and determine where to store each of them. This is usually a trade-off between predictable memory accesses, which are possible when the given structure is stored in the local PLM, and size of the accelerator, which can be reduced by storing more data structures in DRAM. Then, the design of the private local memories of an accelerator comes with many other decisions such as how many memory blocks, what sizes should the blocks be, how can they be arranged to reduce resource usage, and what bandwidth should be used. It is time-consuming for designers to make these decisions manually, so they increasingly rely on automated design space exploration methods to predict or estimate the effects of their decisions. For example, the *gem5-Aladdin* simulator can be used to explore the design space and inform many memory design choices, such as to use private local memories, scratch pads with DMA or caches, or the local memory size and bandwidth. FPGA prototyping with *ESP* can be used to explore and evaluate full systems before ASIC implementations (Mantovani et al. 2020).

The cost of memory elements has a significant impact on hardware resources, limiting the amount of data that can be stored on-chip. However, if two data structures have nonoverlapping lifetimes, the physical memory banks used to store these structures can be shared as they will be never accessed concurrently. Similarly, if it can be guaranteed that two data structures, while they may exist at the same time, are never written concurrently or read concurrently, these structures can be placed

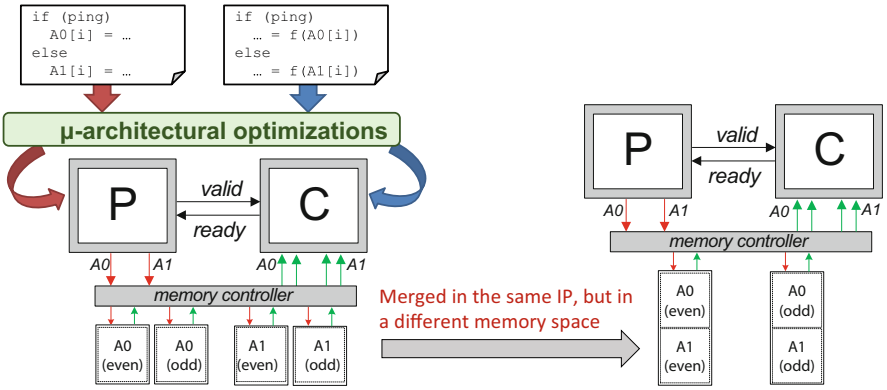


Fig. 8 Resource optimizations for memory IPs with *Mnemosyne* (Pilato et al. 2017)

in the same memory bank (but in different memory spaces), and there will not be port contention. This memory bank sharing can greatly reduce the resource usage. *Mnemosyne* is a tool designed to generate an optimized memory architecture based on a set of characteristics of the data structures and the access patterns that can be provided by the designers (Pilato et al. 2017). The tool analyzes such compatibilities and applies automatic technology-aware transformations (supporting both FPGA and ASIC technologies) to select the proper physical banks in the given technology and determines how to share these physical banks when the data structures mapped on them have disjoint lifetimes. Mnemosyne encapsulates the memory banks in lightweight memory interfaces that translate the logical requests to the data structures into physical requests to the generated bank configuration (see Fig. 8).

Irregular memory accesses are often data dependent or require dynamic resolution of the pointers. These operations are complex when executed in hardware because of the limited flexibility, and only few HLS tools support memory architectures with **dynamic pointer resolution**. For example, *Bambu* builds a daisy-chain architecture on top of alias-analysis results, as shown in Fig. 9. The memory allocation step *resolves* the pointers, i.e., converts them into classic memory operations, when the set of possible target data structures is limited to one. Such operations are later directly connected to memory that stores the corresponding data structure, potentially increasing the parallelism on memory operations. Operations with pointers that cannot be "resolved" are connected in daisy-chain with the potential memories. At design time, the HLS engine assigns a specific memory address to every data structure and, in turn, the associated memory space both on-chip and off-chip. At run time, when an address is propagated through the daisy-chain, only one memory will be activated by the request. In particular, when the address refers to data allocated off-chip, the request reaches the external memory interface and is sent to the corresponding memory controller. The scheduling phase can further distribute the memory accesses to hide the latency of memory transfers especially when accessing the data off-chip.

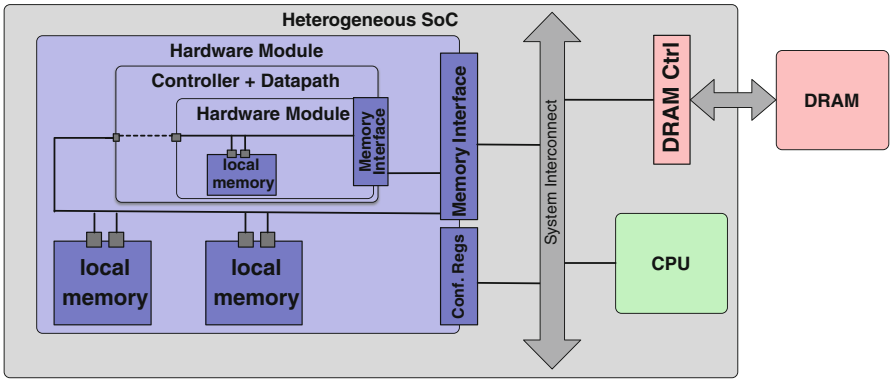


Fig. 9 Daisy-chain memory architecture to support the dynamic resolution of memory addresses in *Bambu* (Pilato et al. 2011b; Pilato and Ferrandi 2013)

Latency-insensitive protocols enable the creation of **latency-tolerant architectures**, called *elastic circuits* (Josipovic et al. 2017a) or *dynamically scheduled architectures* (Minutoli et al. 2016). These architectures can parallelize memory accesses in a way similar to CPU out-of-order execution of the instructions (Josipovic et al. 2017b) but may have high hardware costs.

Multithreaded software using libraries such as pthreads or OpenMP can be synthesized into parallel hardware (Choi et al. 2017). However, parallel hardware often attempts to access the same memory at the same time, resulting in contention and delay. To hide the latency of memory requests and reduce the resource usage of the "hardware threads," designers can interleave the hardware execution of different functions as done in software threads when resources are busy (Hsiao and Anderson 2019). Each hardware thread can have its own private memory with a banked architecture.

Memory allocation techniques can map data onto private local memories and caches based on the dynamic memory requirements. Designers propose **overlay** components to increase the flexibility of the FPGA-based systems. For example, *Intel* recently proposed an FPGA overlay for neural network (NN) inference that can support different NN architectures without reconfiguring the logic cells but changing only the configuration of the component. Supporting these components requires additional HLS compilation steps to generate the overlay configurations as a set of software-like microinstructions. These configurable layers can also help isolate the computation and the outstanding memory requests for security reasons.

Other flexible approaches include **systolic array architectures**. These architectures are becoming popular for deep neural networks, thanks to their flexibility and scalability. *Gemmini* is a representative example of such architectures (Genc et al. 2019). It includes an array of "simple" *processing elements*, which perform MAC operations and rounding bitshifts. The PE array can be configured statically or dynamically to execute different dataflows. The surrounding memory architecture enables full utilization of the MAC units and can be customized with information

coming from HLS. When performing operations on the PE array, the data are moved from the main memory to the on-chip memories, and the array is configured to execute the given dataflow. Ping-pong buffers are used to overlap computation and communication. The same optimizations described above (multi-port memories and distribution of the accesses to avoid conflicts) can be applied to optimize the architecture. Designers must explore similar trade-offs between more banks for higher throughput and fewer banks for better wiring and physical constraints.

Creation of the FSM Controller

After defining the complete microarchitecture of the accelerator, the HLS engine must define the control part, i.e., the component that determines the control signals for the datapath in each clock cycle. This part is modeled as a deterministic **finite-state machine** (FSM). An FSM is a directed graph where each node represents a *control state* and the edges are the transitions from one state to another. Each control state contains the set of operations to be executed in the corresponding clock cycle, along with the control signals for the datapath resources (e.g., multiplexer selectors and register write-enable signals). For each operation to execute on a given functional unit, the FSM determines the paths for providing the input values to the functional unit and the results to the next resource (either another functional unit in chaining or a target register). Based on the scheduling and the evaluation of the conditions in the datapath, the control state determines which transition to activate, i.e., which is the next control state to execute.

The controller FSM manages not only the execution inside the accelerator but also the **system-level synchronization** with external components. This aspect is relevant especially for control-dominated applications (where the accelerator's execution can vary based on external signals) and streaming architectures (where data availability can change the accelerator dynamics). In these cases, the controller must receive additional control signals from the rest of the system to determine which specific transitions must be activated inside the FSM.

RTL Generation and System Integration

After the HLS engine defines the microarchitecture of the accelerator (both datapath and controller), the last phase produces the HDL descriptions for the subsequent logic synthesis and the auxiliary files for system-level integration.

Code Generation, Evaluation, and Verification

Code generation produces common HDL languages like Verilog, SystemVerilog, and VHDL. Some components may require **target-dependent descriptions** to be compatible with synthesis tools. For example, *Bambu* generates slightly different descriptions of the internal memories when targeting different FPGA vendors.

FPGA synthesis tools may infer these components in different ways. Similarly, targeting ASIC technologies requires to instantiate the vendor-specific descriptions of the proper Static RAMs. For example, *Mnemosyne* provides an abstraction to low-level details. The designer is only required to provide a wrapper around the specific SRAMs to match the standardized signal descriptions.

In this phase, many tools (e.g., *Xilinx Vivado HLS* and *Bambu*) provide **early estimations** on the hardware cost of the design. While complete synthesis can provide more accurate results, this process is time-consuming and not feasible when the designer needs to explore many alternatives before finding the most favorable implementation. These estimations are based on different methods, including cumulative costs of single resources, linear regressions, and graph neural networks to include feedback from actual synthesis steps (Makrani et al. 2019).

Once the design has been finalized, the designers also need to verify that the hardware execution produces the expected results. Indeed, errors can be caused by incorrect language specifications, misuse of synthesis directives, wrong connection of the components, or bugs in the HLS tools. While formal verification is a well-established step in logic and physical synthesis, this approach requires reference and current designs to be described in hardware languages. However, in case of HLS, the input code is usually a software-level, untimed specification. Therefore, simulation-based approaches remain the preferred solution for **HLS verification** (Chen et al. 2017). Heterogeneous architectures exacerbate these verification issues for the application programmers, especially when the computation is distributed across software and hardware tasks, and the accelerators are generated with different methods (e.g., preexisting IPs, manually designed components, and HLS modules created with different toolchains). Hardware/software debugging allows designers to backtrack the origin of a bug to the exact point of failure. Since the complexity of architectures is increasing, bugs could be exercised after a long execution time. Therefore, in case of system-level verification, simulation-based approaches have been progressively replaced by on-chip debugging. Many FPGA vendors, for example, provide automated methods to trace internal signals, exposing them to the user to identify execution anomalies. Advanced methods automatically identify discrepancies between hardware execution and the expected behavior (precomputed in software), restricting the area where the error originated (Fezzardi et al. 2015). These approaches are particularly efficient in FPGA, thanks to the reconfiguration of the logic. The designer can implement a design with on-chip monitors, execute it directly on the target system to verify the behavior, and modify the functionality in case of problems. For this reason, **FPGA prototyping** is largely used also in ASIC design flows to emulate the functionality of the entire chip and test the interaction with the software, including the operating system (OS) (Mantovani et al. 2016b).

System-Level Integration and Optimization

The HLS-generated components require integration with other **preexisting components** and the rest of the system. For example, accelerators that interact with

off-chip memory require an interface to the **physical memory controller**. Many vendors provide synthesizable IPs that hide the specific details of the controllers, exposing a standard interface to the accelerators. Such interfaces are usually based on standardized protocols, like AXI4, OpenCAPI, or Wishbone.

Hardware/software integration requires the proper **software stack** to be designed to invoke the accelerator from the software code and exchange data with it. The ESP platform is a paradigmatic example of seamless integration of HLS-generated components with a standard operating system (OS) (Mantovani et al. 2020). From the application viewpoint, the user interacts with the accelerator through two functions for memory allocation and deallocation, and one function to start the accelerator's execution. At a lower level, the interaction is based on standard OS device drivers that are automatically generated during the accelerator design flow. The drivers perform I/O operations on memory-mapped registers to configure the input ports of the accelerators and control the execution. An accelerator's termination signal is connected to a standard interrupt line. When the accelerator completes its execution, it triggers the interrupt routine that reads the output results from the shared memory or the I/O registers. This software stack simplifies the integration of accelerators, minimizing the changes required to the original application.

Domain-specific libraries can provide abstractions to efficiently create FPGA systems. For example, *Xilinx Vitis AI* provides optimized IP cores, tools, libraries, and models to deploy AI algorithms on the Xilinx FPGA devices. Such abstractions include interfaces to machine-learning frameworks like *Caffe* and *TensorFlow*, internal model representations for efficient HLS implementations, specific AI optimizations (e.g., model quantization), runtime libraries, and configuration overlays for the Deep Learning Processing Unit (DPU), which is a configurable engine optimized for convolutional neural network modules.

Open and Modern Challenges

This section discusses open challenges that still need to be addressed in HLS and modern challenges in hardware design that can be efficiently addressed and tackled with the support of HLS.

Creation of Domain-Specific Architectures

So far, most of the optimizations focused on improving the computational part of an accelerator to extract more parallelism and reduce the execution latency, leaving the memory latency to dominate in the overall execution time. DRAM speed is not increasing with the cumulative speed of all components (especially in the case of parallel and/or heterogeneous computing), leading to the so-called memory wall: the overall performance becomes limited by the latency of the memory accesses. To address this **memory bandwidth bottleneck**, specialized and carefully crafted memory architectures can help to hide or reduce the latency when accessing the data.

However, this process is complex as it requires several details that are currently lost in the compilation process. Software programmers usually need little to no thought to be put into memory management, where hardware optimizations (e.g., complex cache hierarchies, bypassing, etc.) hide the latency. In case of accelerators, the specialization of the memory architectures adds effort to the development time, both at the software and hardware levels (Dally et al. 2020). At the *software level*, algorithms need to be reworked to reap the full benefits of a custom memory architecture. Domain-specific languages, like Spatial, incorporate hardware abstractions to ease the description of common memory operations and speed up the development of these accelerators. For example, operations on arrays, tensors, and matrices are common in scientific and machine-learning applications. Such operations must be translated into efficient operations regardless of the allocation of the data and the location/implementation of the memories. At the *hardware level*, deciding how to partition a software data structure requires many considerations. On one hand, larger memories are slower, use more power, and use more resources. On the other hand, the external routing and address calculation logic for a collection of many smaller memories have the same issues. A middle ground between one large memory and many small memories exists which minimizes these issues, but finding the right compromise can be difficult. Automated high-level synthesis can be useful to perform rapid design space exploration to find a desired trade-off point. Specialized modules for pre-fetching the data from external memories can hide the communication latency but require global information on the use of the data. In case of systolic array architectures, the designers have to carefully define the local memories and the buffers between them and the processing elements to also coordinate the data transfers with the off-chip memory. However, this process requires more information than is available in current HLS flows.

Emerging technologies will play a key role in the design of efficient, secure, and reliable accelerators. First, integrated voltage regulators will enable fine-grained power management with dynamic voltage-frequency scaling, while dual-rail memories will help reduce the static power consumption. In addition, some applications, like graph analytics, require frequent but irregular memory accesses to off-chip memory, whose I/O circuitry and refresh activities are responsible for up to 30% of the system energy consumption (**power wall**). The specialization of memory architectures, in this case, must include **emerging memory technologies**. However, many novel technologies, like Hybrid Memory Cube and High Bandwidth Memory technologies, are 3D solutions that can store only few gigabytes of data, while DDR4 can store up to hundreds of gigabytes (**capacity wall**). Architectures that combine DRAM with *nonvolatile memory technologies* (NVM), which require no refresh, can store up to terabytes of data but require a careful co-design that involves the entire stack (Hameed et al. 2018).

Finally, all specializations require modifications to the HLS input languages to embed more domain-specific information. Domain-specific languages (DSLs) will be used by application engineers to specify the algorithmic core of the computation. They will be increasingly used to provide rich information to the compiler and lower-level tools about the high-level semantics of the algorithms.

While DSLs can help describe functional requirements (like the operations among data structures), additional annotations allow designers to express nonfunctional requirements or constraints. For example, *Bambu* already supports the specification of custom data allocation via XML files to specialize the creation of the accelerator's memory architecture and the simplification of the logic to compute the addresses. However, a co-design of the accelerators with the algorithm description would allow the automatic configurations of these steps (see Section "Programmability and System-Level Optimization"). However, DSLs are usually hard to be accepted by software programmers and they may create integration issues. An interesting approach is to use DSL only for specific application kernels embedded in traditional languages, e.g., with template meta-programming in C++, or for high-level specifications of specific workloads (e.g., machine-learning algorithms) (see ▶ Chap. 28, "FPGA-Specific Compilers" for more details).

Programmability and System-Level Optimization

Hardware/software integration has open challenges for **data allocation** in the case of large data sets. Current solutions allow users to allocate data at the software side and transparently access them from the accelerators; these solutions are processor-centric. Data are allocated in the memory hierarchy of the processors (see, e.g., the Intel HARP prototype system), or the accelerator requires efficient methods not only to manage the data locally but also to hide the latency to access them. The design of domain-specific accelerators requires the approach to be changed and the components to be co-designed with the domain-specific memory architecture and the corresponding allocation policy. To do so, designers need a unified representation of the software code and the HLS specification to apply holistic transformations at both sides. Modern compiler representations, like MLIR (Lattner et al. 2021), are gaining attraction but require specific customization within HLS frameworks. These representations will enable and ease the integration of specific **memory-specific transformations** in the front-end phase. For example, loop transformations will be coordinated with transformations in the data structures to improve the data accesses based on the technology of the memories and the location of the data. Rich information about the data access patterns of the algorithm allows the compiler to extract more parallelism and embed more intelligence in the memory controllers.

Finally, **design space exploration** will become more and more important to offer alternative solutions to the designers. Indeed, a fully automated approach is almost impossible to achieve also because the effects of compiler optimizations and transformations are often application-dependent. Therefore, HLS tools have limited view or control on the synthesis process and no well-established flows or sequence of passes. Designers must apply transformations, analyze and understand the results, and derive knowledge for further optimization. This process is the same as in software compilers for code optimization, where machine-learning approaches have been proposed for compiler autotuning (Ashouri et al. 2018).

Hardware Security and Data Protection

FPGA-based systems are typically used in applications and algorithms, like machine learning, that are rapidly changing. The flexibility of FPGA systems is also used by cloud service providers to reuse the resources across multiple users (*tenants*). However, this opens up the possibility that malicious providers can copy the design's intellectual property or users can steal sensitive data of other applications (Pilato et al. 2018a). While protection methods exist for specific cases, their manual application becomes unfeasible for large systems due to their cost or for nonexpert designers due to their complexity. Also, the heterogeneity of the system can introduce new vulnerabilities due to the interaction of components that are not designed at the same time. For example, accelerators have fixed functionality; therefore, it is not possible to perform *code injection*. However, a malicious attacker can launch **software-based attacks** by providing configuration parameters or system configurations that exploit known vulnerabilities in single components or their interactions.

Physical attacks can exploit the weaknesses of a given hardware implementation for stealing private data (*information leakage*). For example, side-channel attacks can extract secret data by analyzing nonfunctional effects (like power consumption and timing characteristics) of the device execution. Accelerators can mitigate side-channel attacks by scrambling the execution to make a uniform power consumption or ensuring constant execution time to thwart timing attacks (Jiang et al. 2018). The modular HLS flow can accommodate additional passes to automatically integrate these extensions and co-optimize them with the accelerator logic.

Another critical issue is **IP theft**: designing heterogeneous systems is a complex and expensive process. The outcome should be protected from reverse engineering and unauthorized copying, which can create billions of dollars of economic damages for the semiconductor design houses. While designers are more sensitive to this problem for ASIC, it is finding an increasing interest also in the FPGA and HLS community. First, designers must guarantee that outsourcing the execution of their designs to third-party cloud providers does not leak details about their algorithms or implementations. Then, embedded FPGAs are also used in several integrated circuits to host specific functions to hide, where HLS identifies and implements such functions to fit into the given logic (Chen et al. 2020). Such security features, like watermarking and obfuscation, can be introduced on the top of the HLS results (Pilato et al. 2018b,c). For example, TAO applies semantic obfuscation during HLS to the design of an accelerator that is able to thwart reverse engineering.

Conclusion

Thanks to the possibility of reusing logic resources across multiple applications and users, FPGA-based systems are becoming a de facto standard to implement rapidly changing workloads, like modern machine-learning applications. High-level

synthesis is a key enabling technology for the creation of such systems. Nonexpert designers can use HLS to create specialized accelerators directly from high-level specifications, focusing only on the algorithmic development. HLS then creates and optimizes the hardware components based on the user's requirements hiding most of the effort from the designers. This chapter provided an overview on the HLS process, describing the existing approaches for the different HLS phases: the analysis of the input specification, the creation of the accelerator microarchitecture, and the generation of the output files. It also discussed open challenges, like the creation of domain-specific architectures and the programmability issues, and modern challenges, like security concerns, that can be addressed with the support of HLS.

References

Ashouri AH, Killian W, Cavazos J, Palermo G, Silvano C (2018) A survey on compiler autotuning using machine learning. ACM Comput Surv 51(5):1–42

Bazargan K, Kastner R, Ogrenci S, Sarrafzadeh M (2000) A c to hardware/software compiler. In: Proceedings of the IEEE symposium on field-programmable custom computing machines (FCCM), pp 331–332

Bombieri N, Liu H-Y, Fummi F, Carloni LP (2013) A method to abstract RTL IP blocks into C++ code and enable high-level synthesis. In: Proceedings of the ACM/EDAC/IEEE design automation conference (DAC), pp 1–9

Brewer F, Gajski DD (1990) Chippe: a system for constraint driven behavioral synthesis. IEEE Trans Comput-Aided Des Integr Circuits Syst 9(7):681–695

Brisk P, Kaplan A, Sarrafzadeh M (2004) Area-efficient instruction set synthesis for reconfigurable system-on-chip designs. In: Proceedings of the ACM/EDAC/IEEE design automation conference (DAC), pp 395–400

Brisk P, Dabiri F, Jafari R, Sarrafzadeh M (2006) Optimal register sharing for high-level synthesis of SSA form programs. IEEE Trans Comput-Aided Des Integr Circuits Syst 25(5):772–779

Buyukkurt B, Cortes J, Villarreal J, Najjar WA (2011) Impact of high-level transformations within the ROCCC framework. ACM Trans Archit Code Optim (TACO) 7(4):17

Canis A, Choi J, Aldham M, Zhang V, Kammoona A, Czajkowski T, Brown SD, Anderson JH (2013) Legup: an open-source high-level synthesis tool for FPGA-based processor/accelerator systems. ACM Trans Embed Comput Syst (TECS) 13(2):24:1–24:27

Canis A, Brown SD, Anderson JH (2014) Modulo SDC scheduling with recurrence minimization in high-level synthesis. In: Proceedings of the IEEE international conference on field programmable logic and applications (FPL), pp 1–8

Carloni LP (2015) From latency-insensitive design to communication-based system-level design. Proc IEEE 103(11):2133–2151

Chang E-S, Gajski DD, Narayan S (1996) An optimal clock period selection method based on slack minimization criteria. ACM Trans Des Autom Electron Syst (TODAES) 1(3):352–370

Chatarasi P, Neuendorffer S, Bayliss S, Vissers K, Sarkar V (2020) Vyasa: a high-performance vectorizing compiler for tensor convolutions on the Xilinx AI engine

Chen D, Cong J (2004) Register binding and port assignment for multiplexer optimization. In: Proceeding of the Asia and South Pacific design automation conference (ASPDAC), pp 68–73

Chen W, Ray S, Bhadra J, Abadir M, Wang L (2017) Challenges and trends in modern SoC design verification. IEEE Des Test 34(5):7–22

Chen J, Zaman M, Makris Y, Blanton RDS, Mitra S, Schafer BC (2020) DECOY: DEflection-Driven HLS-Based Computation Partitioning for Obfuscating Intellectual Property. In: Proceedings of the ACM/IEEE design automation conference (DAC), pp 1–6

Choi J, Brown SD, Anderson JH (2017) From pthreads to multicore hardware systems in legup high-level synthesis for FPGAs. IEEE Trans Very Large Scale Integr (VLSI) Syst 25(10): 2867–2880

Cilardo A, Flich J, Gagliardi M, Gavila RT (2015) Customizable heterogeneous acceleration for tomorrow's high-performance computing. In: Proceedings of the IEEE international conference on high performance computing and communications (HPCC), pp 1181–1185

Cong J (2015) High-level synthesis and beyond – from datacenters to IoTs. In: Proceedings of the IEEE international system-on-chip conference (SOCC), pp 1–1

Cong J, Zhang Z (2006) An efficient and versatile scheduling algorithm based on SDC formulation. In: Proceedings of the ACM/EDAC/IEEE design automation conference (DAC), pp 433–438

Cong J, Fan Y, Han G, Jiang W, Zhang Z (2006) Platform-based behavior-level and system-level synthesis. In: Proceedings of the IEEE international SOC conference, pp 199–202

Cong J, Liu B, Neuendorffer S, Noguera J, Vissers K, Zhang Z (2011) High-level synthesis for FPGAs: from prototyping to deployment. IEEE Trans Comput-Aided Des Integr Circuits Syst 30(4):473–491

Cong J, Huang M, Pan P, Wang Y, Zhang P (2016) Source-to-source optimization for HLS, pp 137–163

Cota EG, Mantovani P, Guglielmo GD, Carloni LP (2015) An analysis of accelerator coupling in heterogeneous architectures. In: Proceedings of the ACM/EDAC/IEEE design automation conference (DAC)

Coussy P, Chavet C, Bomel P, Heller D, Senn E, Martin E (2008) GAUT: a high-level synthesis tool for DSP applications, pp 147–169

Dai S, Liu G, Zhang Z (2018) A scalable approach to exact resource-constrained scheduling based on a joint SDC and SAT formulation. In: Proceedings of the ACM/SIGDA international symposium on field programmable gate arrays (FPGA), pp 137–146

Dally WJ, Turakhia Y, Han S (2020) Domain-specific hardware accelerators. Commun ACM 63 (7):48–57. ISSN 0001-0782

De Micheli G (1993) High-level synthesis of digital circuits. In: Advances in computers, vol 37. Elsevier, The Netherlands, pp 207–283

de Fine Licht J, Besta M, Meierhans S, Hoefler T (2021) Transformations of high-level synthesis codes for high-performance computing. IEEE Trans Parallel Distrib Syst 32(05):1014–1029

Edwards SA (2006) The challenges of synthesizing hardware from c-like languages. IEEE Des Test 23(5):375–386

Edwards SA, Townsend R, Barker M, Kim MA (2019) Compositional dataflow circuits. ACM Trans Embed Comput Syst (TECS) 18(1):1–27

Ernst D, Kim NS, Das S, Pant S, Rao R, Pham T, Ziesler C, Blaauw D, Austin T, Flautner K, Mudge T (2003) Razor: a low-power pipeline based on circuit-level timing speculation. In: Proceedings of the annual IEEE/ACM international symposium on microarchitecture (MICRO), pp 7–18

Esmaeilzadeh H, Blem E, St. Amant R, Sankaralingam K, Burger D (2012) Dark silicon and the end of multicore scaling. IEEE Micro 32(3):122–134

Fezzardi P, Castellana M, Ferrandi F (2015) Trace-based automated logical debugging for high-level synthesis generated circuits. In: Proceedings of the IEEE international conference on computer design (ICCD), pp 251–258

Gajski DD (1984) Silicon compilers and expert systems for VLSI. In: Proceedings of the ACM/EDAC/IEEE design automation conference (DAC), pp 86–87

Galuzzi C, Panainte EM, Yankova Y, Bertels K, Vassiliadis S (2006) Automatic selection of application-specific instruction-set extensions. In: Proceedings of the IEEE/ACM/IFIP international conference on hardware/software codesign and system synthesis (CODES+ISSS), pp 160–165

Genc H, Haj-Ali A, Iyer V, Amid A, Mao H, Wright J, Schmidt C, Zhao J, Ou A, Banister M, Shao YS, Nikolic B, Stoica I, Asanovic K (2019) Gemmini: an agile systolic array generator enabling systematic evaluations of deep-learning architectures. arXiv preprint arXiv:1911.09925

Giri D, Chiu KL, Di Guglielmo G, Mantovani P, Carloni LP (2020) ESP4ML: platform-based design of systems-on-chip for embedded machine learning. In: Proceedings of the ACM/EDAC/IEEE design, automation & test conference in Europe (DATE), pp 1049–1054

Guglielmo GD, Pilato C, Carloni LP (2014) A design methodology for compositional high-level synthesis of communication-centric SoCs. In: Proceedings of the ACM/EDAC/IEEE design automation conference (DAC), pp 1–6

Gupta S, Dutt N, Gupta R, Nicolau A (2003) Spark: a high-level synthesis framework for applying parallelizing compiler transformations. In: Proceedings of the international conference on VLSI design, pp 461–466

Hadjis S, Canis A, Sobue R, Hara-Azumi Y, Tomiyama H, Anderson JH (2015) Profiling-driven multi-cycling in FPGA high-level synthesis. In: Proceedings of the ACM/EDAC/IEEE design, automation & test conference in Europe (DATE), pp 31–36

Hameed F, Khan AA, Castrillon J (2018) Performance and energy-efficient design of STT-RAM last-level cache. IEEE Trans Very Large Scale Integr (VLSI) Syst 26(6):1059–1072

Horowitz M (2014) 1.1 computing's energy problem (and what we can do about it). In: Proceedings of the IEEE international solid-state circuits conference digest of technical papers (ISSCC), pp 10–14

Hsiao H, Anderson JH (2019) Thread weaving: static resource scheduling for multithreaded high-level synthesis. In: Proceedings of the ACM/EDAC/IEEE design automation conference (DAC)

Huang Q, Lian R, Canis A, Choi J, Xi R, Calagar N, Brown SD, Anderson JH (2015) The effect of compiler optimizations on high-level synthesis-generated hardware. ACM Trans Reconfig Technol Syst (TRETS) 8(3):14:1–14:26

Jiang Z, Dai S, Suh GE, Zhang Z (2018) High-level synthesis with timing-sensitive information flow enforcement. In: Proceedings of the IEEE/ACM international conference on computer-aided design (ICCAD), pp 1–8

Josipovic L, Brisk P, Ienne P (2017a) From c to elastic circuits. In: Proceedings of the asilomar conference on signals, systems, and computers (ACSSC), pp 121–125

Josipovic L, Brisk P, Ienne P (2017b) An out-of-order load-store queue for spatial computing. In: Proceedings of the IEEE symposium on field-programmable custom computing machines (FCCM), pp 134–134

Josipović L, Ghosal R, Ienne P (2018) Dynamically scheduled high-level synthesis. In: Proceedings of the ACM/SIGDA international symposium on field programmable gate arrays (FPGA), pp 127–136

Klimovic A, Anderson JH (2013) Bitwidth-optimized hardware accelerators with software fallback. In: Proceedings of the IEEE international conference on field-programmable technology (FPT), pp 136–143

Koeplinger D, Feldman M, Prabhakar R, Zhang Y, Hadjis S, Fiszel R, Zhao T, Nardi L, Pedram A, Kozyrakis C, Olukotun K (2018) Spatial: a language and compiler for application accelerators. In: Proceedings of the ACM SIGPLAN conference on programming language design and implementation (PLDI), pp 296–311. ISBN 9781450356985

Kotsifakou M, Srivastava P, Sinclair MD, Komuravelli R, Adve V, Adve S (2018) HPVM: heterogeneous parallel virtual machine. In: Proceedings of the ACM SIGPLAN symposium on principles and practice of parallel programming (PPoPP), pp 68–80

Ku DC, Micheli GD (1991) Constrained resource sharing and conflict resolution in hebe. Integration 12(2):131–165

Lattner C, Amini M, Bondhugula U, Cohen A, Davis A, Pienaar J, Riddle R, Shpeisman T, Vasilache N, Zinenko O (2020) Mlir: a compiler infrastructure for the end of Moore's law

Lattner C, Amini M, Bondhugula U, Cohen A, Davis A, Pienaar J, Riddle R, Shpeisman T, Vasilache N, Zinenko O (2021) MLIR: scaling compiler infrastructure for domain specific computation. In: Proceedings of the IEEE/ACM international symposium on code generation and optimization (CGO), pp 2–14

Lattuada M, Ferrandi F (2015) Code transformations based on speculative SDC scheduling. In: Proceedings of the IEEE/ACM international conference on computer-aided design (ICCAD), pp 71–77

Lattuada M, Ferrandi F (2019) A design flow engine for the support of customized dynamic high level synthesis flows. ACM Trans Reconfig Technol Syst (TRETS) 12(4):1–26

Liu J, Cong J (2019) Dataflow systolic array implementations of matrix decomposition using high level synthesis. In: Proceedings of the ACM/SIGDA international symposium on field programmable gate arrays (FPGA), p 187

Makrani HM, Sayadi H, Mohsenin T, Rafatirad S, Sasan A, Homayoun H (2019) XPPE: cross-platform performance estimation of hardware accelerators using machine learning. In: Proceedings of the 24th Asia and South Pacific design automation conference (ASPDAC)

Mantovani P, Cota EG, Pilato C, Guglielmo GD, Carloni LP (2016a) Handling large data sets for high-performance embedded applications in heterogeneous systems-on-chip. In: Proceedings of the international conference on compliers, architectures, and sythesis of embedded systems (CASES), pp 3:1–3:10

Mantovani P, Cota EG, Tien K, Pilato C, Guglielmo GD, Shepard K, Carloni LP (2016b) An FPGA-based infrastructure for fine-grained DVFS analysis in high-performance embedded systems. In: Proceedings of the ACM/EDAC/IEEE design automation conference (DAC)

Mantovani P, Giri D, Di Guglielmo G, Piccolboni L, Zuckerman J, Cota EG, Petracca M, Pilato C, Carloni LP (2020) Agile SoC development with open ESP. In: Proceedings of the IEEE/ACM international conference on computer-aided design (ICCAD), pp 1–6

Martin G, Smith G (2009) High-level synthesis: past, present, and future. IEEE Des Test Comput 26(4):18–25

Minutoli M, Castellana VG, Tumeo A, Ferrandi F (2015) Inter-procedural resource sharing in high level synthesis through function proxies. In: Proceedings of the IEEE international conference on field programmable logic and applications (FPL), pp 1–8

Minutoli M, Castellana VG, Tumeo A, Lattuada M, Ferrandi F (2016) Efficient synthesis of graph methods: a dynamically scheduled architecture. In: Proceedings of the IEEE/ACM international conference on computer-aided design (ICCAD)

Nane R, Sima VM, Pilato C, Choi J, Fort B, Canis A, Chen YT, Hsiao H, Brown S, Ferrandi F, Anderson J, Bertels K (2016) A survey and evaluation of FPGA high-level synthesis tools. IEEE Trans Comput-Aided Des Integr Circuits Syst 35(10):1591–1604

Ndu G (2012) Boosting single thread performance in mobile processors using reconfigurable acceleration. PhD thesis, 10

Pilato C (2017) Bridging the gap between software and hardware designers using high-level synthesis. In: Proceedings of the international conference on parallel computing (PARCO), pp 622–631

Pilato C, Ferrandi F (2013) Bambu: a modular framework for the high level synthesis of memory-intensive applications. In: Proceedings of the IEEE international conference on field programmable logic and applications (FPL), pp 1–4

Pilato C, Tumeo A, Palermo G, Ferrandi F, Lanzi PL, Sciuto D (2008) Improving evolutionary exploration to area-time optimization of FPGA designs. J Syst Archit Embed Syst Des 54(11): 1046–1057

Pilato C, Castellana VG, Lovergine S, Ferrandi F (2011a) A runtime adaptive controller for supporting hardware components with variable latency. In: Proceedings of the NASA/ESA conference on adaptive hardware and systems (AHS), pp 153–160

Pilato C, Ferrandi F, Sciuto D (2011b) A design methodology to implement memory accesses in high-level synthesis. In: Proceedings of the IEEE/ACM/IFIP international conference on hardware/software codesign and system synthesis (CODES+ISSS), pp 49–58

Pilato C, Mantovani P, Guglielmo GD, Carloni LP (2017) System-level optimization of accelerator local memory for heterogeneous systems-on-chip. IEEE Trans Comput-Aided Des Integr Circuits Syst 36(3):435–448

Pilato C, Garg S, Wu K, Karri R, Regazzoni F (2018a) Securing hardware accelerators: a new challenge for high-level synthesis. IEEE Embed Syst Lett 10(3):77–80

Pilato C, Basu K, Shayan M, Regazzoni F, Karri R (2018b) High-level synthesis of benevolent trojans. In: Proceedings of the ACM/EDAC/IEEE design, automation & test conference in Europe (DATE), pp 1124–1129

Pilato C, Regazzoni F, Karri R, Garg S (2018c) TAO: techniques for algorithm-level obfuscation during high-level synthesis. In: Proceedings of the ACM/EDAC/IEEE design automation conference (DAC), pp 1–6

Pilato C, Wu K, Garg S, Karri R, Regazzoni F (2019) TaintHLS: high-level synthesis for dynamic information flow tracking. IEEE Trans Comput-Aided Des Integr Circuits Syst 38(5):798–808

Pilato C, Bohm S, Brocheton F, Castrillon J, Cevasco R, Cima V, Cmar R, Diamantopoulos D, Ferrandi F, Martinovic J, Palermo G, Paolino M, Parodi A, Pittaluga L, Raho D, Regazzoni F, Slaninova K, Hagleitner C (2021) EVEREST: a design environment for extreme-scale big data analytics on heterogeneous platforms. In: Proceedings of the design, automation, and test in Europe conference and exhibition (DATE)

Pothineni N, Brisk P, Ienne P, Kumar A, Paul K (2010) A high-level synthesis flow for custom instruction set extensions for application-specific processors. In: Proceedings of the IEEE Asian and South Pacific design automation conference (ASP-DAC), pp 707–712

Pu J, Bell S, Yang X, Setter J, Richardson S, Ragan-Kelley J, Horowitz M (2017) Programming heterogeneous systems from an image processing DSL. ACM Trans Archit Code Optim 14 (3):1–25

Ragan-Kelley J, Barnes C, Adams A, Paris S, Durand F, Amarasinghe S (2013) Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In: Proceedings of the ACM SIGPLAN conference on programming language design and implementation (PLDI), pp 519–530. ISBN 9781450320146

Ranjan Panda P, Dutt ND, Nicolau A (1998) Incorporating dram access modes into high-level synthesis. IEEE Trans Comput-Aided Des Integr Circuits Syst 17(2):96–109

Stok L (1994) Data path synthesis. Integration 18(1):1–71

Venkatesan R, Shao YS, Wang M, Clemons J, Dai S, Fojtik M, Keller B, Klinefelter A, Pinckney N, Raina P, Zhang Y, Zimmer B, Dally WJ, Emer J, Keckler SW, Khailany B (2019) Magnet: a modular accelerator generator for neural networks. In: Proceedings of the IEEE/ACM international conference on computer-aided design (ICCAD), pp 1–8

Weerasinghe J, Polig R, Abel F, Hagleitner C (2016) Network-attached FPGAs for data center applications. In: Proceedings of the international conference on field-programmable technology (FPT), pp 36–43

Windh S, Ma X, Halstead RJ, Budhkar P, Luna Z, Hussaini O, Najjar WA (2015) High-level language tools for reconfigurable computing. Proc IEEE 103(3):390–408

Zhu J, Gajski DD (1999) A unified formal model of ISA and FSMD. In: Proceedings of the seventh international workshop on hardware/software codesign (CODES), pp 121–125

Processor Simulation and Characterization

25

Grant Edmund Martin, Suhas Madhusudana, Greg Efland, and Vadim Kustov

Contents

Introduction . . . 876
Application and Algorithm Analysis . . . 879
Data Types and Operations . . . 879
Algorithms . . . 880
Example: Affine Transform of 2D Image . . . 881
New or Existing Processor? . . . 882
Existing Processor . . . 882
Extending Configurable Processor . . . 883
New Processor with New ISA . . . 883
Hybrid Mode: New ISA with Custom Extensions . . . 884
Standard Benchmarks . . . 884
Issues with Estimating Processor Performance . . . 884
Whetstone . . . 888
Linpack . . . 888
Dhrystone . . . 888
CoreMark . . . 889
Embench . . . 890
SPEC CPU . . . 891
EEMBC . . . 892
Berkeley Design Technology . . . 892
Summary . . . 892
Using Application Code for Benchmarking . . . 893
Estimation Analysis . . . 893
Examples of Estimation Flow . . . 895
For Further Consideration . . . 901

G. E. Martin (✉)
Pleasanton, CA, USA
e-mail: gmartin@ieee.org

S. Madhusudana · G. Efland · V. Kustov
Cadence Design Systems, Tensilica R&D, San Jose, CA, USA
e-mail: suhasm@cadence.com; gefland@cadence.com; vkustov@cadence.com

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_20

Processor Simulation 902
Functional Simulation 902
Cycle-Level Simulation 905
Hardware Emulation 907
Using Processor Simulators in System Modelling 907
Summary Table Comparing Various CPU Modelling Abstractions 909
Examples 909
Conclusion 911
References 911

Abstract

Performance characterization and analysis of processors have two main purposes. The first is to optimize the design of a new processor. The second is to make a choice among various processors for a particular design project. A new processor might have a new instruction set architecture (ISA), but more likely will create a derivative of a configurable, extensible processor. To support this process, processor characterization and analysis, using simulation models and benchmarks are essential. Benchmarks may be industry standards with years of development and use, or application-oriented code drawn from the design target domain.

This chapter will outline the various use cases and history of processor characterization approaches and benchmarks. It will also discuss the construction and use of processor simulation. It uses as examples methods available for specific families of configurable, extensible processors, as well as other references drawn from industry and academia, to illustrate the principles in real use.

Keywords

Processor simulators · Benchmarks · Processor characterization · Processor customization and extensibility

Introduction

Design of electronic systems in the third decade of the twenty-first century almost inevitably requires a design team to choose one or more embedded processors, potentially of various kinds (CPU, DSP, GPU, specialized application processors), as part of the design decomposition and functionality mapping (often known as hardware-software codesign). The world offers a huge variety of choices to design teams despite the consolidation among processor instruction set architectures (ISAs) that has occurred over the past two decades. Teams may want to design a brand-new processor with a brand-new ISA, although this has become less and less common. Teams may be constrained to choose among existing processor implementations, either at the full chip/system-on-chip (SOC) level with fully packaged processors, or as intellectual property (IP) blocks already predetermined as a design constraint.

On the other hand, teams may be able to use configurable and extensible processor technology to create derivatives of existing processor ISAs: they may choose coarse-grained processor parameters or even add new application-oriented instructions to the ISA to fit the design needs. For example, they may tune the sizes of caches and local tightly coupled memories to ensure that a deeply embedded processor has just enough memory for its applications, but no more. If their application emphasizes floating point operations, they may turn on an IEEE 754 single-precision floating-point ISA option. If they are using a preconfigured neural net acceleration processor, they may have proprietary NN models which would benefit greatly in performance and power by adding proprietary accelerating instructions using an architectural description language (ADL).

Whether choosing among constrained existing alternatives or given more latitude in choosing to design a brand-new processor or creating a derivative using configurable technology, it is vital to take a data-driven approach to making these design decisions. Characterizing design alternatives for the optimal fit to the design requirements is a vital step. Where a great deal is known about the intended application space, the best results are found when drawing from existing or developing new application-oriented benchmarks. Where the application space is so general purpose that it is unclear what the best measurement criteria are to use for judgment, standard benchmarks may represent the only reasonable alternative, although even there, the choice of which benchmark(s) to use is important. Many benchmarks have been proposed and used for the last several decades, and they have risen into and fallen out of favor. Even where application-oriented benchmarks are available or newly written, it may be important to characterize the processor choices using standard benchmarks because high performance on them, even if not particularly relevant to the use cases, may be necessary for promoting the resulting product to end design teams or to demonstrate some level of "future-proofing" by showing high general purpose performance.

For existing designs, especially if implemented in packaged chips, running the benchmarks may be relatively straightforward and design choices easy to justify. For IP choices, especially with newly configured processor ISAs, use of various models is necessary and their fidelity to the ultimate physical expression is an important criterion in justifying such choices.

Having extracted a variety of benchmark data across the range of credible processor choices, it is important that a design team have a defensible analysis method to allow optimal choices to be made while considering the various aspects of performance, power, and area (PPA), and execution speed estimates derived from the models. Objective numerical criteria must be supplemented by more subjective qualitative criteria to justify the choice ultimately made, and it is important to use consensus weightings for the criteria to justify the decisions made.

The need to justify decisions rests partly on the tools used to measure performance, including processor simulators which exist at wide variety of levels. The ▸ Chap. 26, "Methodologies for Design Space Exploration" by Andy Pimentel outlines the importance of processor simulation in design space exploration and summarizes several levels, including:

- RTL level
- Cycle-accurate instruction set simulation (ISS)
- ISS using binary translation (functional level, sometimes regarded as emulators)
- Host-compiled simulation where the functional ISS approach is combined with the target application code
- Trace-based simulation as opposed to the previous execution-based approaches
- Sampled simulation
- Statistical simulation

And, not using simulation at all, analytical models. In addition, processor simulators are used in system level modelling environments and tools, sometimes called virtual platforms or virtual prototypes (see ▸ Chap. 27, "Virtual Prototyping of Processor-Based Platforms" by Tim Kogel).

The structure of this chapter is perhaps a little unusual, in that it is based on a design flow that a team may use to select and optimize processors for their application. By describing the design flow in more detail, we will illustrate the methodologies used for processor simulation and characterization. The design flow consists of several steps as illustrated in Fig. 1.

We start with deciding on the key applications that are most important for the target design domain for which the processor(s) are intended. Next, we analyze these applications for their key data type requirements and the key algorithms

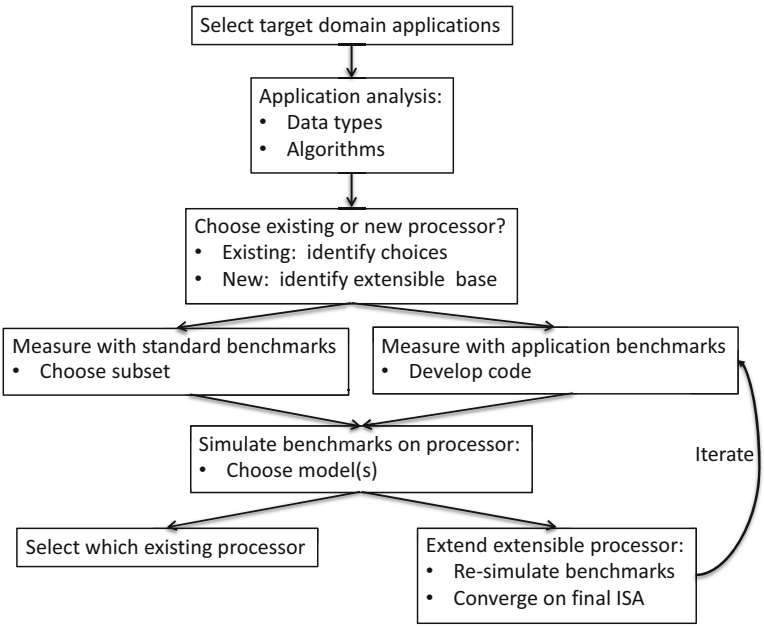


Fig. 1 Processor selection and characterization flow

which determine the performance of the applications. The next step is to choose whether this design project will use existing processors or SoC devices with already determined processor choices, or whether we could create new processors, probably using a commercial or noncommercial extensible processor as a base.

Depending on the nature of the design project, we might choose to decide on processors using standard benchmarks, or we may choose using application-oriented benchmarks for which we may need to develop the code for it. It is possible that we may use a combination of both types of benchmarks. Measurements are derived using appropriate models of the processor choices, of which there are several types as discussed. The chapter will detail the several types of models that are possible.

Using the measurements, if the design project is using existing processors as a design space, then a final choice is possible. If an extensible processor is being used, then there is an interesting iterative loop in which the ISA may be extended and the application performance is remeasured using updated models, which should eventually terminate in a well-defined extended, configured processor which will be used in the final design.

Finally, prior to the conclusion, we will illustrate several of these concepts using examples of configurable and extensible processors with a wide variety of application domains (audio, imaging and computer vision, communications, sensor and signal processing, and AI/ML) as the basis for a discussion. A set of useful references ends the chapter.

Application and Algorithm Analysis

The starting point for performance characterization is always the intended applications. These may be existing or newly developed applications. Although we open with data types rather than algorithms, in fact these are heavily interrelated. A high-level algorithm may be chosen for the application, then the data type requirements for accuracy and precision studied, and then the details of the algorithm fleshed out to accommodate the data type characteristics.

Data Types and Operations

Applications are often written initially as a reference implementation that focuses on the clarity of the implementation and performance of the algorithm itself (e.g., the overall error rate of a soft decoder).

Reference implementations often use data types convenient for algorithm development. They might use floating point, for example, where the dynamic range of values is not well understood initially. Algorithms captured in modelling tools such as MATLAB (Mathworks 2020) use native double-precision floating point as a default data type; within the tools, refinement to fixed-point or other floating-point representations (such as single-precision, IEEE 754 half-precision, or newer 16-bit floating-point types such as bfloat16) may be possible.

Whether done directly in a modelling tool or as a specific stage in the algorithmic analysis and transformation process, mapping the reference implementation to an efficient embedded implementation optimizes the data types to the application requirements. This might include replacing 32-bit integers with more efficient types supported by the programming language such as 16-bit or 8-bit integers for an existing processor or creating new types such as 4-bit integers for an extensible processor. Analysis of the algorithm requirements and performance characteristics of the target processor need to influence the choices. Trade-offs include memory, throughput, and power and energy requirements.

Existing applications may have been optimized for a specific target processor including the data types and operations. This may place requirements on processor selection. In some cases, re-optimizing for a new target processor may be necessary to meet performance requirements.

Algorithms

Algorithms are closely linked to the data types and operations they use. In fact, different algorithms may be necessary to best exploit the operations and data types available to a given processor.

For example, consider implementing the affine transform on a two-dimensional (2D) image (Wolberg 1990). A common approach is to interpolate the value of the corresponding pixel in the input image for each position in the output image using the inverse mapping. This can be done directly as a 2D interpolation or with a two-pass approach of one-dimensional (1D) interpolations. The former can be efficient on a vector processor when gather instructions are provided, while the latter 1D approach may be more efficient when gather instructions either are not available or have limited throughput.

Understanding the target applications and their requisite algorithms and data types is key to the selection of the benchmarks used for processor performance analysis. Existing benchmarks may be similar to or match the target but use different algorithms or data types and thus not be representative of actual performance. If no existing benchmark can be found, it may be necessary to create one.

Analysis of algorithm complexity is often useful to judge feasibility and guide processor selection. Algorithm complexity can include types and numbers of operations (e.g., number of 16-bit fixed-point multiply-accumulates (MACs)) and memory sizes. This can also be used to evaluate benchmark results on a given processor – for example, what utilization of the processor resources is achieved.

As an example, one can imagine a controller which controls the position of a wide-angle camera and communicates with the user, who controls the camera via the controller. Once frames from the camera are captured and written to a memory accessible by the controller, the captured raw frames may not be convenient for viewing by a human observer, because the viewing position of the camera undergoes high-speed changes as the car moves. Without some form of digital image stabilization, the output video may look jerky and distorted.

One of the ways to perform digital image stabilization is to apply affine transforms on the raw images before they are displayed.

Example: Affine Transform of 2D Image

An affine transform of an image is a two-dimensional transform. The image after the affine transform contains values at locations (x′, y′) which are derived from values of the source image (n, m) through the following formula:

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} af_{11} & af_{12} & bf_1 \\ af_{21} & af_{22} & bf_2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} n \\ m \\ 1 \end{pmatrix}$$

The transformation is two dimensional represented by the submatrix af_{nm} (f stands for forward transform) and the third dimension with parameters bf_n represents a translation. (n, m) are coordinates of the pixel in the digital image before the transformation and are integer or natural numbers, and (x′, y′) are new coordinates of the value at (n, m). (x′, y′) are not guaranteed to be integers and commonly are real numbers.

In practice, an inverse procedure is preferred where the inverse transform is performed to get the input coordinates. In this case, the output coordinates (x′, y′) are integer or natural numbers, and the corresponding input coordinates (n, m) are generally real numbers. Since we are dealing with sampled digital images, the value of the input at coordinates (n, m) must be interpolated.

Consider the following pseudo-code implementation of the affine transform of a 2D image.

Listing 1 Affine transform of a 2D image pseudo-code

```
for y 0..H-1
  for x 0..W-1
    # compute corresponding position in source image (xs,ys)
    float xs = ar11*x + ar12*y + ar13;
    float ys = ar21*x + ar22*y + ar23;

    # nearest neighbors in source image
    xsi = int(xs); xsf = frac(xs);
    ysi = int(ys); ysf = frac(ys);
    tl = srcimg[xsi][ysi  ]; tr = srcimg[xsi+1][ysi  ];
    bl = srcimg[xsi][ysi+1]; br = srcimg[xsi+1][ysi+1];

    # interpolate output value
    t = tl + (tr-tl)*xsf;
    b = bl + (br-bl)*xsf;
    dstimg[x][y] = t + (b-t)*ysf;
```

Here the inverse mapping is used to determine the corresponding position of each output pixel in the input image. The value of the corresponding input pixel at this position is interpolated, in this example using simple bilinear interpolation of the four nearest neighboring pixel values.

This description uses floating-point values to represent the mapping parameters and the corresponding input positions, and to perform the interpolation. While this may be an appropriate choice for a processor with good support for floating-point types and operations, it may be better to use fixed-point types on processors without hardware floating-point support. In this case, one needs to determine the range and precision requirements based on image sizes and interpolation quality to determine the appropriate fixed-point formats.

The complexity of the transform kernel includes several steps. The first step is evaluation of the inverse mapping to compute the corresponding position in the source image. As written, this requires four multiplications and four additions. However, these positions can be incrementally calculated using simple additions for each iteration of the nested loops. This reduces the complexity to two additions in the inner loop and two in the outer loop.

The next step is to compute the positions of the nearest neighbors and read them from the source image. This includes two operations to separate the positions into integer and fractional components. Reading the source image values requires additions of a constant – often this can be done with an addressing mode.

The final step is to interpolate the value given the values of the four neighbors and write the result to the destination image. As written, the interpolation requires three multiplications and six additions. Note, however, the difference input to each of the multiplications requires a larger range to represent than the individual pixel values and may be problematic; an alternative is to use two separate multiplications for each instead. Some processors may provide fused difference-multiply instructions or even full interpolation instructions to optimize handling of such cases.

Algorithm complexity analysis is useful for estimating the first-order computation requirements of an application as a guide for initial processor selection and evaluation. The specifics of a processor will be considered in subsequent analysis and benchmarking.

New or Existing Processor?

In a nutshell, there are four prominent use cases that are important in choosing processors.

Existing Processor

Choosing existing and stable processors, possibly as packaged parts (or portions of packaged SoCs which will be used for an application). For example, Mediatek and Qualcomm both offer packaged chips which include application processors for

mobile devices. Both use variants of Arm cores and are available in multiprocessor configurations (Mediatek 2020, Qualcomm 2020). For a mobile device design, an OEM may wish to choose the provider of their application processor SoC and then choose which among their various offerings is the best combination of price, performance, future capacity, power consumption, and ease of programming, for example. While performance is not the only criterion to use, it is certainly an important one and thus characterizing the variety of application processors on offer to help find the best match to the mobile product requirements is an essential use case.

It is also quite possible that a design team is planning to develop their own SoC but will want to choose one of several preconfigured instruction set processors available as intellectual property from different vendors (e.g., Arm, Cadence-Tensilica, Ceva, Synopsys-ARC), without wanting to make any particular extensions, and may make modest configuration choices such as memory sizes. In this case, they can be regarded in almost the same way as prepackaged, pre-built SoCs, albeit with slightly more variation possible in configuring some of the design characteristics. Again, choice must be made among vendors and then among possible configurations available from the chose vendor, and various characteristics will be important, of which performance is only one.

Extending Configurable Processor

Choosing processors from commercial IP suppliers with configurable, and/or extensible, proprietary processor ISAs and instantiations. For example, Arm, Cadence-Tensilica, Ceva, Synopsys-ARC are all possible suppliers. Here there is both a major supplier choice to be made and then a more complex choice of the best processor configuration to use for the application. This is made more complex when the ISA can itself be extended to provide better application specificity and performance, power, and area characteristics. Examples later in this chapter will illustrate how processor configurability and extensibility intersect with processor characterization and benchmarks to allow optimal choices to be made. Processor extension is usually accomplished by technology supporting an architectural description language (ADL) (Ienne and Leupers 2007, and also ▸ Chap. 23, "Architecture Description Languages" by Anupam Chattopadhyay, Z. Wang and G. Martin).

New Processor with New ISA

The option of designing a new processor, with a new instruction set architecture (ISA), has declined in favor over time. This is due to the high costs of developing a processor design and implementation from scratch, verifying it, and developing and supporting the complete software (SW) toolchain for it – (compiler, assembler, disassembler, debugger, IDE, instruction set simulator(s)). However, there are commercial ADL-based toolsets that make this option easier: for example, Synopsis

ASIP Designer, based on the nml ADL (Synopsys 2021); and Codasip's Codasip Studio, using the CodAL ADL (Codasip 2021).

Hybrid Mode: New ISA with Custom Extensions

However, over the last several years, new options have arisen that reflect a hybrid between a brand-new ISA and choosing only existing processor implementations. RISC-V, for example, (RISC-V 2020) allows a design team to pick a well-defined base ISA with some well-defined configurable additions (growing over time) and then extend it further with proprietary custom ISA, while still benefiting from third party, often open source, SW tooling. In addition, there are a variety of third party, often open source, but also commercial, IP offerings in the RISC-V domain (RISC-V Exchange 2020), which can be used as-is, configured and extended by the end user group, or by a commercial supplier, thus offering a credible hybrid model for design groups interested in this approach.

Standard Benchmarks

Before going into detail on the available standard benchmarks and their use cases, we will discuss generally how to estimate and measure processor performance as an aid to making processor selection, and the general issues with estimating processor performance. This is used to motivate the use of standard benchmarks, and we describe several used historically and more recently. In many cases, the standard benchmark numbers may be sufficient for making processor choices; in other cases, moving to application-specific benchmarks may be needed to make the right choice. This is the topic for the next section.

Issues with Estimating Processor Performance

About two decades ago, it was common for processor vendors to advertise the performance of their products by stating how many instructions per second the processor could execute. Some customers, especially start-up companies, based their choice of the processor on this criterion alone, only to find later that processors capable of executing fewer instructions per second (commonly measured in millions of instructions per second or MIPS) outperformed processors capable of more MIPS on their tasks. Nowadays, MIPS is still looked upon as the first step in evaluating processor choice or design, but MIPS is almost never used as a single performance measure, because it does not distinguish between the types of instructions and architectures of the processors.

To illustrate the problem, let us consider a task of subtracting two images. This task is common in video compression (such as MPEG-2, H.264, H.265) where a motion-compensated picture is subtracted from the original uncompressed picture

to form a residual signal, which is then quantized and encoded. If the pictures are more than 8 bits per pixel, and processor A has an ISA that only has an 8-bit subtract instruction, processor B has a 32-bit subtract instruction, and processor C has an instruction that performs multiple 32-bit subtractions in one cycle, it will take the smallest number of instructions to subtract the pictures using processor C, more instructions on processor B, and the highest number of instructions on processor A. Processor A may be capable of the highest number of MIPS (for example because it can operate at a higher frequency), while processor C will perform the task of subtracting images faster because it will need to execute fewer instructions.

Often processor vendors characterize the performance not in instructions per second but operations per second. This metric will consider processor C's capability of performing multiple operations within a single instruction, and multiply the number of operations in the instruction by the number of instructions per second (typically measured in giga or tera operations per seconds: GOPs or TOPs). This metric gives a better insight into the processor's resources, but still falls short of predicting performance on a specific task, since it is a measure of computational resources, not of architectural fit. As an example, one can imagine two machines both capable of executing one GOP, the first machine executing 1000 single operation MIPS, the other 100 MIPS, each a 10-way single instruction multiple data (SIMD) instruction. If otherwise the architectures of the two machines are similar, they will have a similar performance in subtracting two images. However, given an algorithm with high data dependencies, such as entropy coding, it will be difficult to schedule 10 operations in parallel, and the computational resources of the SIMD machine will be underutilized giving the machine with a single operation per instruction running at a higher frequency a performance advantage.

Modern applications, such as video compression, robotic vision, audio compression, graphics, and neural networks, in their entirety are far more complicated than subtracting two images or entropy coding. Choosing the right architecture for specific applications is an extremely important task. Even if a processor meets the computational budget to perform the task, it can be an overkill for the task resulting in idle resources, increased power consumption, area, and cost.

Several decades ago, professionals developing processor hardware as well as software architects who used the processors started to look at a unified way to characterize performance of processors, searching for applications (benchmarks) which would better predict processor performance for a wide variety of tasks than MIPS or GOPs.

Let us look at some C language code snippets to highlight the problem looking at the code. The first example is a common procedure of linked list parsing. We will look at the inner loop where most algorithm implementations spend most of the processor time.

```
struct Node
       {void  *data;
        struct Node *next;
       };
struct Node* node;
```

```
Instantiate_and_initialise_Node_struct (node);
```

And the inner loop will look like:

```
while (node != NULL)
    {
        process(node->data);
        node = node->next;
    }
```

Here *process()* is some abstract function assumed to be fast compared to accessing data: *node->data* and addressing the next node *node->next.* There is a dependency between accessing the current address and finding the new address. Even if the hardware architecture allows for multiple simultaneous memory accesses, the algorithmic dependency on accessing the current address to compute the next address will not utilize multiple memory accesses. That does not mean that all processors will have the same performance on this code. Processors with a short (e.g., one cycle) memory read latency would tend to perform better than processors with a longer memory read time.

In the second code snippet, we will look at a different scenario, where there is no algorithmic data dependency between inner loop iterations. The code snippet implements a simple alpha blending of two images. Note that *output_image[m][n]=image1[m][n]*alpha+(1-alpha)*image2[m][n]* are independent from *output_image[i][j]=image1[i][j]*alpha+(1-alpha)*image2[i][j]* as we go through the loop, as long as the output image, image1 and image2 arrays are not aliased. If the compiler knows that the addresses are not aliased and SIMD additions and multiplications as long as wide load/stores to keep MACs busy are present, the compiler may map multiple iterations of the inner loop into fewer or just one iteration(s), processing multiple indices n in a single SIMD instruction. This is an example where the number of MACs and the load/store width would matter.

Listing 2 Blending two images

```
for (h=0;h<height;h++)
       for (w=0;w<width;w++)
              output_image[h][w]=image1[h][w]*alpha+
(1-alpha)*image2[h][w];
```

Instead of statically counting processor resources, benchmark code is executed on the target machine, the execution time is measured (either the wall clock time or normalized cycle count "per Megahertz"), and the speed of code execution is reported.

Thus, a benchmark considers not only computational resources, but underutilization and stalls due to data dependencies, memory access latencies, and other architectural features.

The drawback of benchmarking is that any benchmark has a finite number of instruction patterns or code and can still fail to predict the target machine performance on a specific application or a class of applications. A benchmark that does predict performance for any application would be "universal," but it does not exist.

Benchmarks can be characterized by the domain where they do a good job of predicting, e.g., general CPU, signal processing, graphics, neural nets, etc.

In this section, we will focus our discussion and analysis on commonly used benchmarks which can be obtained and used for benchmarking purposes at no cost, while touching on some commercial benchmarks. We will concentrate on benchmarking general purpose CPUs which exclude GPUs and application-targeted DSPs (e.g., DSPs with ISA optimized for audio processing). The delineation between high-end embedded controllers and general-purpose DSPs is becoming blurred. General purpose DSPs, which traditionally outperformed controllers in computational power, started to feature full MMUs and security features which were the features of high-end controllers, while high-end controllers have added more computational performance (commonly through use of SIMD operations and a wide adoption of superscalar architecture), which was the purview of DSPs. For example, Arm Neon or Helium (Marsh 2020) can be used as a controller or as a DSP, users of Cadence-Tensilica controllers can add custom instructions increasing their DSP capability, RISC-V core architecture is intended for use as a controller/CPU, but RISC-V with vector extension as a DSP.

The oldest pair of benchmarks which are commonly used to this day are Whetstone and Dhrystone, both intended for general purpose computing performance testing. One reason for that is around the time Whetstone (in the 1970s) and Dhrystone (in the 1980s) were developed, embedded systems were very simple from a programming and functional point of view. Microcontrollers executed small programs written in assembly or proprietary machine-specific low-level languages. For very simple programs and simple architectures, pencil-and-paper approaches to estimating microcontroller performance provided adequate accuracy. So, during this period (1970s and 1980s), the benchmarking targets were primarily CPUs in desktops and servers.

In the 1990s, the industry saw the proliferation of DSPs and embedded controllers, programmable in higher-level languages such as C and C++. In terms of the target architectures, benchmarks like CoreMark started refocusing on this space, even though Whetstone and more commonly Dhrystone were still used.

Later multicore embedded systems with complex memory topologies and interaction architectures became mainstream. Whetstone, Dhrystone, and CoreMark are not designed to measure interaction and efficiencies of multicore systems and have several deficiencies in measuring single core performance. In addition, newer benchmarks such as EEMBC were not free and thus not easily available to academic researchers or small start-ups, especially with the fostering of a new generation of ISA research enabled by the RISC-V movement (RISC-V 2020). To fill that gap, a new suite of benchmarks, Embench, is gaining traction as of 2020.

Whetstone

The first version of Whetstone (Curnow and Wichmann 1976) was developed in 1972 and in its modified form is still used today. In its current form, it is a small amount of synthetic code which contains loops, function calls, fixed- and floating-point computations. In this context, "synthetic" means it does not implement algorithms or applications but contains artificial code. Although Whetstone attempts to include different types of computation and memory accesses as a "universal" benchmark would do, the benchmark is heavily weighted by mathematical computation using floating point, and branches. To achieve a good performance on Whetstone, the processor must have double-precision floating-point operations in its instruction set. A good branch predictor and branch resolution architecture will help achieve higher performance on Whetstone. For general purpose application processors which are intended to run different programs, fetch large amounts of instructions (Whetstone instruction size is small and can fit into a modern level one instruction cache), perform frequent context switching, performance on the Whetstone benchmark does not correlate well with real-world performance. However, if the intended applications are floating DSP-type applications, Whetstone is useful at an early stage of selecting a processor, although so outmoded as of 2020 that its use is rarely reported.

Linpack

In 1979, another floating-point benchmark called Linpack became available which is still used today. The goal of the benchmark has never been to estimate the performance of general-purpose processors on a blend of typical general-purpose algorithms. The benchmark performs an LU decomposition of a matrix in floating point (double-precision 64-bit computations) and has been thought to be representative of scientific computation. It has been used extensively for benchmarking supercomputers. This benchmark is mentioned briefly here since the focus of this chapter is general purpose benchmarks. For more information, see Dongarra (Dongara et al. 2003).

Dhrystone

The first version of Dhrystone was developed in 1984 (version 1.1). Just like Whetstone, this is a synthetic benchmark and does not represent any real-world application. Just like Whetstone, the code contains loops, branches, function calls, and fixed-point computations, but no floating-point computations. Dhrystone was designed to better represent execution patterns and types of computation encountered in general purpose processors. There are several drawbacks in Dhrystone

which stand in the way of obtaining realistic performance estimates of real-world applications on general purpose processors:

(a) The outputs of some functions are not used, and compilers in the optimization stage would remove those functions as dead code, so they will not be executed at all, which works against the intention of the Dhrystone benchmark, giving extremely good and incorrect Dhrystone scores.
(b) Dhrystone contains too much string manipulation code, so the score is heavily weighted by the ability to manipulate strings. String manipulation is not correlated well with the types of computations and memory accesses where general purpose computers and controllers spend most of their computational performance today. Performance on string manipulations do not correlate well with general purpose DSPs and controllers.
(c) Dhrystone's use of libraries makes it heavily dependent on library optimizations, and therefore more dependent on the compiler which skews the benchmarking of the underlying processor hardware.
(d) Just like Whetstone, Dhrystone is a small program and could fit completely into instruction caches of modern processors. Therefore, Dhrystone does not well characterize the instruction caching subsystem, the effect of the system bus, and external memory bandwidth and latency.

Subsequent versions of Dhrystone (2.0 and 2.1) attempted to modify the code to prevent dead code elimination by the compiler. The effort, however, was only partially successful. To obtain a meaningful score, one must disable dead code elimination typically through compiler optimization flags.

In 2020, Dhrystone is still used as a measure of performance primarily in controller-type processors, although even for these types of applications, there are better and newer benchmarks. For the past two decades, the consensus has been that Dhrystone is on its way out, and over the next couple of years, it would be uncommon to ask for or provide Dhrystone scores. Nonetheless, as of 2020, Dhrystone is still often used. For more information, see Weiss (2002).

CoreMark

CoreMark (EEMBC Coremark 2020) was developed in 2009 to measure processor performance in embedded systems. It was intended to address the drawbacks of Dhrystone and replace it. CoreMark addressed several shortcomings of Dhrystone.

Every operation in the timed part of the code is computed during runtime and cannot be precomputed during compile time. CRC checks the correctness of the operations; it also creates a dependency which the compiler cannot resolve. Thus, due to this code structure, the compiler cannot optimize out a portion of the code, as is the case with Dhrystone.

CoreMark does not contain string manipulations. In fact, it does not make library calls within the timed part of the code. This makes it independent of library optimizations and makes it more of a hardware benchmark than a library benchmark.

Like Dhrystone, it is a synthetic benchmark which does not include floating-point computations. It contains integer arithmetic, matrix manipulations, linked-list parsing, state machines, data-dependent conditional branches, and CRC computations.

Like Whetstone and Dhrystone, the instruction size and data size are also small on most target processors, so this benchmark could fit completely into instruction caches or data caches of modern processors. As far as instruction cache is concerned, the situation here is somewhat "worse" than that of Whetstone and Dhrystone. This is because without library calls, the amount of code which gets cached is less than that of Whetstone and Dhrystone where library code gets cached as well.

Therefore, like Dhrystone and Whetstone, it is not useful for the characterization of the caching subsystem, the effect of the system bus, and external memories' bandwidths and latencies.

CoreMark is a good benchmark for general purpose and controller types of processors. For digital signal processing, it is not an adequate benchmark, because it does not perform mathematical operations such as multiply-accumulate, representative of such processing, nor does it contain a sufficient amount of independent data and operations which can be exploited through parallel data and instruction processing (VLIW and/or SIMD). Its lack of floating point makes it unsuitable for processors which emphasize floating point capabilities.

Overall CoreMark is a big improvement for estimating the performance of embedded controllers. Currently it has gained enough traction to be considered a successor of Dhrystone. However, processor vendors and their customers may still want Dhrystone numbers, most of the time in addition to, and not instead of CoreMark. EEMBC has released a cost-free version of Coremark on Github (Coremark Github 2020).

Embench

Rounding up the short list of the most common no-cost benchmarks is Embench. Unlike previously discussed benchmarks, Embench is not a single benchmark but a suite of different benchmarks. It is still under development, as of the time of this writing in late 2020. In fact, Embench's philosophy is to adapt to new situations and benchmarking new architectures over time as the need arises (Embench 2020; Patterson 2020). To quote from its website as to is purpose:

> Dhrystone and Coremark have been the de facto standard microcontroller benchmark suites for the last thirty years, but these benchmarks no longer reflect the needs of modern embedded systems. Embench™ was explicitly designed to meet the requirements of modern connected embedded systems. The benchmarks are relevant, portable, and well implemented.

The goal of the Embench suite is to have about 20 or so kernels representing a variety of computational patterns encountered in real-world applications (avoiding

synthetic code). The applications are selected to cover three axes of processor architecture: computational power, memory accesses, and branches, and implement diverse algorithms. To illustrate the diversity of the applications in the Embench suite, it is sufficient to name some: 32-bit CRC checking, cubic root solver, Huffman encoding and decoding, dot product, FIR, IIR, DCT, codebook search, integer matrix multiplication, matrix inversion, N-body problem calculations, Regex, mean, standard deviation, correlation, etc. Embench uses fixed-point, single-precision floating-point as well as double-precision floating-point computations. The original code for individual applications comprising the suite was not written specifically for Embench; they come from various sources including earlier benchmarks, but the "pick" of individual applications is what makes the prospect for wide adoption of Embench good.

Finally, it is worth to mention the use of Embench for caching and external memory access performance estimation. Despite the many applications comprising Embench, the total code size is small, moreover applications warm up caches before the timed portion of the code by design, so the suite is not designed to stress the caching system.

SPEC CPU

SPEC CPU (SPEC CPU 2020) stands out among the benchmarks discussed here by its code size. SPEC has large data and code, which are unlikely to fit into Level One (L-1) cache. It would therefore be more likely affected than the other benchmarks by the performance of the caching system and bus and external memory characteristics.

This is a commercial benchmark suite, and there is a cost associated with obtaining the benchmarks. SPEC is a suite of applications which consists of non-synthetic code. The first version of SPEC CPU was made available in 1989 and consisted of 10 programs. The current version of SPEC CPU has 43 benchmarks organized into four suites:

1. SPECspeed®2017 Integer (10 fixed-point benchmarks)
2. SPECspeed®2017 Floating Point (10 floating-point benchmarks)
3. SPECrate®2017 Integer (10 fixed-point benchmarks)
4. SPECrate®2017 Floating Point (13 floating-point benchmarks)

SPECspeed benchmarks execute one copy of a benchmark at a time, while SPECrate benchmarks execute multiple copies of benchmarks.

As was mentioned earlier the benchmarks are large, a single large benchmark can exceed 1 Gbyte in size and therefore would not fit into L-1 caches of modern high clock speed processors. SPEC CPU is a good choice when trying to estimate the performance of the entire system: the core CPU, system buses, different levels of caching, main memory, and nonvolatile memory. Running multiple copies of benchmarks can estimate the performance of multi-threaded multicore systems. This can test how well the OS distributes multiple jobs to multiple cores. In theory, one

copy of SPECspeed can also benefit from multiple threads and cores; however, if the compiler does not support OpenMP, it must be capable of auto-parallelization.

The benchmarks are distributed as source code. To compile all of them, one needs C99, C++2003, and a Fortran-2003 compiler. This may pose a challenge when trying to benchmark embedded or soft cores from smaller processor vendors without a large software ecosystem.

EEMBC

EEMBC (originally, EDN Embedded Microprocessor Benchmark Consortium) (EEMBC 2020) was formed in 1997, sponsored by an electronics trade magazine, EDN (Electronic Design News) to fill a gap in benchmarks available for embedded processors, as opposed to large scientific processing or large servers. The acronym has lived on despite a split from EDN in 2012. These are primarily commercial benchmarks representing application classes and are not free, although EEMBC is a nonprofit and it charges membership and benchmark licensing fees. They have been organized into different classes such as Office Automation (OABench), consumer entertainment (ConsumerBench), Telephony (Telebench), TCP/IP networking (Networking), and automotive (AutoBench) from the early 2000s. Over time, more benchmark suites were added such as DENBench, MultiBench, EnergyBench, CoreMark (in 2009, see above), ULPMark, SecureMark, ADASMark, and MLMark. Some of the benchmarks have been donated to the public domain via Github, such as Coremark, SecureMark-TLS, OABench, and Telebench. Others have been placed in Github for public inspection, such as MLMark and CoreMark-PRO (a successor to CoreMark), but these benchmarks can only be used for publishing results if the user licenses them from EEMBC for this purpose.

For more information on EEMBC, a historical view is presented in Leibson (2006) and Leibson (2016), and their website has current information.

Berkeley Design Technology

Berkeley Design Technology, BDTI (BDTI 2020), is a company which provides multiple services including development of IP, analysis of software, hardware, and trends. Among the services BDTI performs are in-house benchmarking and processor performance comparisons.

Summary

Table 1 contains a summary of the benchmarks discussed in this section.

While standard benchmarks may be useful for making some processor choices, dependent on application, very often the use of application code allows a more optimal choice to be made.

Table 1 Standard benchmarks

	Whetstone	Linpack	Dhrystone	CoreMark (EEMBC)	SPEC CPU	Embench	EEMBC
First version	1972	1979	1984	2009	1989	2020	Founded in 1997
Initial target	Servers, desktops	Supercomputers	Servers, desktops	Embedded	Servers	Embedded	Embedded
Code function	Synthetic	Algorithm	Synthetic	Synthetic algorithms	Applications	Algorithms	Algorithms
Code size	Small	Small	Small	Small	Very large	Small	Large
Cost	Free	Free	Free	Free	$1,000 ($250 academia)	Free	$20,000

Using Application Code for Benchmarking

In the previous sections, we looked at the amount of resources available on the processor and use of common standard benchmarks to predict processor performance. The goal of that section was to describe methods to estimate processor performance on a variety of tasks and thus aid in selecting the right processor or architecture.

In this section, we will look at the problem of estimating processor performance when the applications for which the processor will spend most of its time are known.

In this scenario, the performance of the candidate processor on common standard benchmarks of the previous section is less relevant: our goal is to perform a particular known algorithm on a frame in less than a certain amount of time within a given power budget (and most likely an area budget as well, if this is a chip-level design using a soft-core processor).

Estimation Analysis

As far as the architectural fit and speed of the processor is concerned, the selection of the processor can take place at different levels: selecting a processor based on its fixed architecture, selecting appropriate extension packages to the processor base ISA, and introducing user-defined individual instructions or operations.

Selecting the processor based on its fixed architecture is the coarsest way of obtaining architectural fit and frequently carried out by balancing the amount of available computational resources and bandwidth with the amount required by the algorithm. For example, if the algorithm is MAC limited and requires executing 500 million MACs per second, the processor must be capable of executing at least 500 million MACs per second.

At the next level where the designer extends the processor with a custom ISA package, a more detailed analysis of the algorithm must be carried out to flesh out frequent computational and memory access patterns and try to match them against the ISA of an extension package.

Let us continue with the example of the affine transform from section "Example: Affine transform of 2-D Image." The goal is to sustain a transform of height (H) by width (W) image pixels (for the sake of this example, assume a grayscale image) at a rate of R frames per second. Looking at a standard benchmark, such as a CoreMark score, does not tell us whether we will sustain the transform for this image size and frame rate. Moreover, although we may sustain the needed throughput, the power consumption and area of the core may be an overkill for the job.

As the first step, we need to establish the required computational resource: H x W x R x number of required computations per pixel (Cp). Computations per pixel is not a universal metric; it begs the question of what computations and how they map onto the ISA. Is multiply-accumulate one operation or two operations: multiply and add?

The next question is: are we limited by data dependencies? If the number of required computations per pixel is less than the number of available required computations per pixel, but there is a dependency between computations, they cannot be carried out in parallel.

The next question relates to memory bandwidth. In some cases, the processor can be designed (such as an extension for neural networks) to be capable of many MACs per load, relying on the fact that weights or data are reused and multiple operations are performed on data or weights between reading them from memory and writing the results back to memory. If a high MAC to bandwidth architecture is used to implement affine transforms, the computational blocks could quickly be starved waiting for data to come from memory and then stalled waiting for data to be written to memory.

The question of bandwidth is not just limited to the maximum bandwidth available between the core and the memory subsystem but also to the actual bandwidth limited by the access pattern of the algorithm. The affine transform is a good example where data is not loaded or stored in aligned contiguous patterns. For example, if we can load N bytes of aligned contiguous data from memory, this does not mean that such a performance can be achieved if N bytes are at noncontiguous addresses in memory. If a simple memory management subsystem and ISA are used, the data needs to be loaded using the N-byte word it is contained within. Therefore, if each byte of data belongs to a different N-byte word, to load N bytes of data, we need N N-word loads instead of just one. This might not be a big problem for narrow machines when N is small, for example, 1, 2, or 4 bytes, but it becomes extremely inefficient for wider machines, for example, with N= 64, 128, or more bytes.

If the memory subsystem involves caches, as opposed to tightly coupled memory and direct memory access engine transfers, the analysis becomes much more complicated.

Continuing with the example of the affine transform, our design-oriented approach can be broken into the following steps:

Computational resource requirement: What number and what type of operations per second are needed? Can these operations be executed in parallel? Affine transform is performed on pixels independent of transform on other pixels, therefore it will benefit from wider processors where multiple pixels can be processed in parallel.

Data dependencies: Assuming there are no resource constraints (infinite resource), what is the number of operations which can be scheduled in parallel? We cannot schedule more than that number because the outputs of operations are inputs to other operations. Looking at the pseudo-code of section "Algorithms," the output value of dstimg[x][y] does not depend on any computed value dstimg[x′][y′] and therefore does not have inherent data dependencies. However, an implementation can introduce a dependency: we cannot compute and store dstimg[x][y] until intermediate values t and b have been computed.

Bandwidth: What is the required bandwidth regardless of memory access patterns? If this bandwidth is less than what is available, then we need to consider the access pattern. If the memory access pattern is pseudo-random with small data widths compared to the load/store word width, we need to look at special instructions and HW available on the core which will make loads and stores from/to pseudo-random memory addresses faster. In the affine transform, we see that the input data, in general, is not guaranteed to be at consecutive addresses. This algorithm benefits from wide loads and stores with loads capable of loading from discontinuous addresses in one operation.

Caching: Cache controllers create main memory access patterns and evictions which are complicated in the presence of pseudo-random memory accesses. Tightly coupled or scratchpad memories (TCMs) and direct memory access units (DMAs) are easier to analyze and to come up with the performance prediction. The affine transform is a good example of this. Analysis of DMA-type data transfer model is more straightforward: transfer a block of pixels to TCM, while processing another block of pixels and transferring a block of output pixels from TCM. With appropriate transfer and computational latencies, and block sizes, the transfer time can be completely hidden during the computation time. In a classical cache without prefetch, there will be a guaranteed cache miss when new input pixels are required, which are not in cache. This latency will not be hidden as the processor will stop computations until the input data is available.

Examples of Estimation Flow

Let us consider examples of evaluating RISC-V cores and soft cores from Cadence Tensilica for implementing the affine transform. There are also soft cores such as those from Arm, ARC, CEVA, and MIPS which may be considered, but we limit ourselves to two to get the main points across. Tensilica is representative of commercial extensible processor architectures; it has a mature set of tools for customizing cores as well as creating different purpose DSP ISAs. RISC-V is an open-source standard, which has several extensions either ratified or in definition for

accelerating different algorithms, and many cores available from academic research projects, open-source consortia, and commercial suppliers.

Hardware Aspects

We first start with a configurable, extensible processor as a candidate for implementing affine transform on the video feed described above (Fig. 2).

We start by considering the base processor core which is a 32-bit machine, scalar architecture with one 32-bit instruction executed at a time. If we want a faster floating-point implementation, we need to select a floating-point coprocessor as well. Then we calculate the number of operations identified in the implementation of section "Algorithms" sustained per second against the capability of the base core. If the base core cannot sustain such a load, we can design custom instructions to accelerate the implementation or consider extended DSP cores. Beyond the base Cadence-Tensilica core are families of various application-oriented DSPs including a Vision DSP family. In the Vision family, there are two predefined vision DSPs which give a good architectural fit to this application space. Looking further into the vision DSP family for a resource match, we find that DSP two (Vision Q7) has twice the number of MACs as DSP one (Vision P6), and can run at a higher frequency than DSP one.

Further, if DSP two ISA lacks the required performance, we could consider adding further custom instructions to accelerate our transform. The DSP two ISA is not specifically optimized for affine transforms; rather the ISA is optimized for a blend of video and image processing algorithms and should provide good performance for affine transform of an image. The computational part of its architecture includes the capability of executing up to five SIMD operations in a single VLIW instruction bundle. Each of the five SIMD operations can operate on 512-bit wide inputs, i.e., 64 operations on 8-bit data, 32 operations on 16-bit data

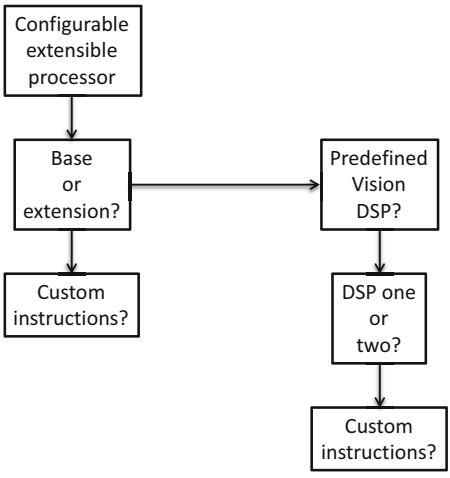


Fig. 2 Extensible processor selection flow

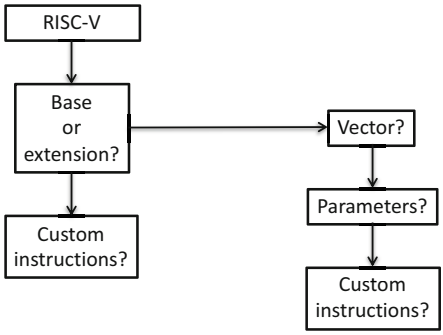


Fig. 3 RISC-V processor selection flow

(including single-precision floating-point operations), 16 operations on 32-bit data, etc. It can sustain 512 8x8 MACs per cycle.

On the memory access side, DSP two sustains two 512-bit aligned loads (TCM to register) or one load and one 512-bit aligned store. It also features gather and scatter operations which take the base address and up to 32 offsets to compute 32 addresses (which are arbitrary) from which the gather operation loads data or to which the scatter operation writes data.

A similar selection methodology can be adopted if we consider RISC-V to implement the transform (Fig. 3).

If the base RISC-V machine does not have enough resources, we may consider extensions. To accelerate an affine transform RISC-V, the proposed RISC-V vector (V) extension is more appropriate for the algorithm.

The RISC-V vector extension, under definition, adds vector instructions to the base ISA. The concept is similar to that of many DSPs. However, even at this early stage of selection, one can see significant differences in architecture which require a different selection flow. Both RISC-V vectors and vision DSPs have 32 logical vector registers for SIMD or vector operations. On the vision DSPs, these logical registers correspond to physical registers 512-bits wide, using an N-way programming model with SIMD types of 2Nx8, Nx16, N_2x32 (where N_2 means N/2). Here the number to the left of "x" represent the number of elements of that type and to the right the width of an element (in bits). For the vision DSPs under consideration, N=32, so 2N=64, N_2 =16.

The RISC-V vector extension defines 32 logical vector registers and vector operations whose width is based on control parameters in special registers. There is no one-to-one correspondence between the logical and physical width of registers and vector operations. RISC-V-vector compliant implementation guarantees that the defined set of vector operations are present in the machine; the specification itself, however, does not define the amount of computational resources or the bandwidth of the implemented core. The core vendor must specify the parameters from which we can compute the available resources.

The vision DSPs feature a 5-slot VLIW instruction, which is present in their definition. The RISC-V ISA does not prescribe VLIW, scalar, in-order superscalar,

or out-of-order instructions, so high-end DSPs may use a super-scalar or out-of-order architecture. Furthermore, the number and types of instructions which a RISC-V implementation can perform in parallel must be defined by the implementor or supplier, and they may offer a family with several alternatives.

As of autumn of 2021, the RISC-V vector extension has not been ratified, although it has reached an advanced state. As with all RISC-V concepts, pre-reserved ISA encoding space allows implementors to add special custom instructions to the selection of RISC-V extensions they are implementing.

By considering the hardware aspects of our application, we have identified some of the key trade-offs and decisions in processor selection and configuration. By diving into the software aspects, we can further refine our analysis and improve our prediction of final application performance.

Software Aspects

Now that we have looked at the hardware aspects of implementing the affine transform, let us look at some of the software aspects.

Section "Example: Affine Transform of 2-D Image" contains an example of an implementation with pseudo-code (Listing 1). That implementation assumes one 32-bit floating-point operation is performed at a time and the number of operations is expressed as the number of 32-bit floating-point operations.

Analyzing the algorithm, we see that computations for every pixel are independent of each other; they can be performed in parallel. We can load multiple pixels, perform multiplication and additions on the loaded pixels in parallel and store them in parallel.

For example, these computations are independent:

$$
\begin{aligned}
&\texttt{xs} = \texttt{ar11}*\texttt{x} + \texttt{ar12}*\texttt{y} + \texttt{ar13}\\
&\texttt{xs}_n = \texttt{ar11}*\texttt{x}_n + \texttt{ar12}*\texttt{y}_n + \texttt{ar13}\ n \in 0..15\ \texttt{or:}\\
&\texttt{vxs} = \texttt{ar11}*\texttt{vx} + \texttt{ar12}*\texttt{vy} + \texttt{ar13}
\end{aligned}
$$

where vxs, vx, and vy are vector or SIMD variables, each containing 16 32-bit single-precision floating point numbers. Vision DSP two contains 16-way 32-bit floating point type as well as two 16-way 32-bit floating point MACs and additions. So, the original line of code can be accelerated up to 32 times.

Loading these pixels in parallel could be tricky. Naturally every processor can load and store pixels which are contiguous and aligned to the word. If the image is aligned on the word boundary of the processor and the image width is a multiple of the word width, every load brings the maximum possible number of pixels. For example, for a 512-bit load, the load operation loads 64 8-bit pixels onto a register or 16 32-bit single-precision floating point numbers. To load multiple bytes or small words from pseudo-random addresses, many wide machines including DSP one and two feature operations designed to perform such operations efficiently. Such operations, found in many DSPs, are called gathers (for loads) and scatters (for stores). The RISC-V vector extension specifies such operations as well. These are loads and stores with the vector-indexed addressing mode.

The basic idea here is that wide memories connected to the machine (for example, 512-bit interfaces) may be constructed from narrower (for example, 32-bit wide) individually addressable memory macros forming 32-bit wide individually addressable memory sub-banks. In the absence of sub-bank conflicts (no more than one access per sub-bank), all sub-banks can be accessed in parallel even though the data items comprising 512 bits could be at pseudo-random discontinuous addresses.

We might further improve the performance by considering that DSP two is a 5-way VLIW machine, and we can perform loads, multiple MACs, and scatter in parallel.

The straightest way of implementing the pseudo-code of section "Algorithms" is to use the corresponding reference C code, give it to the compiler, and expect the compiler to figure out that multiple iterations can be performed in parallel with the number of iterations reduced. This code is for illustrative purposes only; on a real implementation, the programmer or compiler may optimize it further:

```
for x 0..W-1
  # compute corresponding position in source image (xs,ys)
  float xs = a11*x + a12*y + a13
```

will become:

```
for x 0..(W/(N/2))-1
    xb_vecN_2xf32 vxs = a11*vx + a12*vy + a13
```

Here xb_vecN_2xf32 is a notation for a 16-way 32-bit single-precision floating-point C SIMD type. The operators "*" and "+" will be overloaded differently. Instead of single multiplies and multiply accumulates, the compiler will map these operations to vector operations.

Since the vision DSPs have 5-slot VLIW instructions, the compiler will try to schedule up to five operations in parallel, for example:

{vector store, vector load, vector multiply, vector add, nop}.

Here all the vector operations are 512-bit operations or 16 32-bit floating-point multiplies and adds.

In the first part (vectorization), a RISC-V vector compiler will have a similar flow, but the second part, where a VLIW compiler tries to schedule multiple operations into one instruction, does not exist for RISC-V vectors unless a RISC-V vector extension processor uses an underlying VLIW approach. More likely, a RISC-V vector extension processor will use a scalar, in-order superscalar or out of order architecture which may extend to vector operations, and if it is capable of executing more than one instruction in parallel, the schedule will be determined by the machine's hardware dynamically during runtime.

Different algorithms and their reference C implementations present different degree of difficulties for auto-vectorization by the compiler. In many cases, auto-vectorization will fail.

The causes of the compiler's failure to auto-vectorize may be divided into two categories:

1. It is possible to auto-vectorize, but the compiler is unable to establish a mapping from scalar computation to vector computation because the mapping problem is too complex. This is often the case with lane crossing operations and pseudo-random addresses.
2. It is not possible to auto-vectorize based on the information given to the compiler. The code below given to the compiler in isolation looks obvious for auto-vectorization, but this code should fail to auto-vectorize, if the compiler does not have additional information. Knowing that pointers a, b, and c are not aliased, the programmer may go ahead and vectorize the code, but if the compiler does not have this information, auto-vectorization fails.

Listing 3 Vectorization example

```
void process (float *a, float *b, float *c)
{
for (i=0;i<N/2;i++){
a[i]=b[i]*c[i];
}
return;
}
```

In the example with the affine transform, reading pseudo-random locations from memory may pose particular difficulties for auto-vectorization. When auto-vectorization fails, the programmer may attempt to vectorize the code manually, by inserting intrinsics into C code.

```
float a[N/2], b[N/2], c[N/2];
for (i=0; i<N/2; i++) {
  a[i]=b[i]*c[i];
}
```

will become:

```
xb_vecN_2xf32 a, b, c;
a= MULN_2XF32(b, c);
```

Here MULN_2XF32 is a 16-way 32-bit floating-point multiplication operation.

Similar techniques with different names of types and C-intrinsics may be used on a RISC-V vector machine.

For example, similar to DSP two's MULN_2XF32 operation, an operation called vfmul.vv exists in RISCV-vector machines, with the fundamental difference is that the number of processing lanes is not tied to a specific implementation of the RISC-V vector machine. For example, the instruction can tell the processor to perform a 64-way multiplication, but the physical register file width in the implementation might be 256 bits and the number of floating-point multiply units might be 8. In this case, the 64-way multiplication will be performed in 64*32/256=8 steps.

Custom Instructions

As was mentioned before, if the machine still does not meet the computational requirements, custom instructions can be introduced to accelerate cycle performance.

Looking at the pseudo-code of section "Algorithms," we can see that bilinear interpolation could be a candidate for a new operation.

Here is an example:

```
t = tl + (tr-tl)*xsf;
```

We simply create an operation which is equivalent to the above expression by fusing logic.

t=LININTERP(tl, tr, xsf);

Our bilinear interpolation computation

```
t = tl + (tr-tl)*xsf;
b = bl + (br-bl)*xsf;
dstimg[x][y] = t + (b-t)*ysf;
```

now has one-third the number of operations (assuming all variables are contained in registers, no spills, and excluding the final store):

```
t=LININTERP(tl, tr, xsf);
b=LININTERP(bl, br, xsf);
dstimg[x][y] = LININTERP(t, b, ysf);
```

Further we can create a vector variant of `LININTERP`, where we apply this operation to a row of pixels at a time instead of to an individual pixel.

From the physical implementation view, fusing two adds and a multiply is unlikely to decrease the maximum frequency of the machine or require an extra stage for the instruction, if a multiply operation has a tangible positive slack. This means that the reduction of the number of instructions will directly translate to a reduction of the number of cycles and the absolute time the machine will spend on interpolation.

For Further Consideration

In this and the previous section, we have looked at using code as a way of predicting processor performance in the real world while still at an early design stage. Once the methodologies described here have produced an acceptable result, the processor becomes a candidate processor. The next step will be to figure out the requirements on the performance of the subsystem. This includes considering the performance of higher latency memories: L2, L3, etc., performance of the bus to which the core is connected, coherence and synchronization overheads in a multicore subsystem. All these need to be considered – the cost of making a mistake in estimating the performance at this early stage is the most expensive mistake in a project.

Throughout the last two sections we talked about running or executing code at an early stage of a project to determine processors capabilities or a fit. At such an

early stage, silicon is not available. Therefore, one needs some sort of a simulator to benchmark the code. A simulator can also be used to get more visibility into the reasons for performance issues than could be obtained from actual hardware. This is the subject of the next section.

Processor Simulation

In this section, we will discuss commonly used modelling techniques underlying the assessment of algorithm performance on a target processor architecture: simulation and emulation to characterize processor functionality, performance, and efficiency. These modelling techniques vary in speed, accuracy, and configurability, and often require the use of multiple techniques and associated tools at different stages of processor exploration, design, and characterization at the right level of modelling abstraction.

Functional Simulation

Definition

Functional simulation of a processor simply models the functional behavior of its instruction set architecture (ISA). It helps determine if the implementation of a software program is functionally correct. It rarely encompasses any micro-architectural features of the target processor. It merely emulates one instruction at a time by computing the outputs for a given set of inputs. Every simulated instruction is assumed to take one clock cycle to complete. Functional simulators are also quite useful for early architectural simulation as they can generate instruction-level traces, which typically have information that can be consumed by trace-based statistical analysis tools.

Trace-Driven Cache Simulators and Branch-Prediction Simulators

As examples, the simplest forms of cache simulators and branch-prediction simulators are often trace driven. Trace-driven simulators model some micro-architectural features of the target component in isolation and produce useful information both about the program being analyzed – branch percentage, instruction-mix (loads, stores, ALU, MAC, branches, etc.) – and the effectiveness of the component's configuration by simulating its timing (cycle-count performance) characteristics. The obvious benefit of separating functional simulation from the trace-driven timing simulation is that the former is run once, but the latter can be run multiple times for various configurations of the target component. However, a limitation of trace-driven simulation is trace outputs can be prohibitively large for long-running programs, such as SPEC CPU benchmarks. Knowing which parts of the target program to profile is not always easy. For large programs, techniques such as Simpoint (Sherwood et al. 2002) may provide valuable insight into which sections

of those programs have the most impact on overall performance, and thus serve as relevant portions for detailed performance characterization.

Instruction Mix Analysis

Functional simulation can also be used to estimate CPI (cycles per instruction), or, for superscalar architectures, IPC (instructions per cycle), of a processor by starting with an estimate of the number of cycles of latency for each class of instructions – ALU, load, store, branch, MAC, divide, etc., and using this estimate in conjunction with frequency of each class as obtained from simulation-generated trace-based tools. Furthermore, overall CPU time for a program can be estimated knowing total instruction count, processor CPI (estimated above), and clock speed, see formula below.

$$\text{CPU time} = \left(\text{Total instruction count}^{*}\ \text{CPI}\right) / \text{clock speed}$$

For a given processor architecture, CPU time can be reduced by increasing the clock speed, or lowering the CPI, or lowering the program's instruction count, or some combination of them. Applying this analysis to a set of programs with diverse instruction mix (McCallum and Chua 1987) can be a useful exercise in estimating relative performance differences among those programs even with simple functional simulation-based performance models.

Instruction Level Parallelism (ILP)

There are multiple architectural techniques (Misra et al. 2014) to improve overall processor performance by lowering CPI – instruction pipelining, in-order superscalar execution, VLIW execution, vector instructions (for instance, SIMD), out-of-order execution, branch prediction, etc. In addition, compilers too exploit inherent parallelism in programs, for example, transforming loop-level parallelism into instruction-level parallelism (ILP) by unrolling loops. In cases where compilers are aware of underlying hardware, they can take advantage of any special hardware features to improve ILP. While compiler optimizations for a given program can be characterized to the first order using traces from functional simulators by analyzing differences in instruction mix and count, it is not trivial to characterize all the architectural techniques noted above merely by using functional simulators, which do not model the necessary micro-architectural details.

Memory Access Patterns

Locality of memory references is an important factor affecting program performance and analysis of memory access patterns assists in the evaluation of the memory subsystem architecture (Brown et al. 1998). Techniques such as caching and prefetching for memory, and branch predictors for branches improve performance of programs that have a strong locality of reference for memory and branch accesses, respectively. Software optimizations, particularly programming style and compilers, attempt to increase locality of references, while hardware features exploit

the locality of references to minimize time and power spent in accessing data elements in memory.

As an example, for processor architectures that support simultaneous memory access of multiple agents in the same cycle, a popular technique to improve performance is the use of memory banking (Sudarsanam and Malik 1995).

Register-File Usage Analysis

CPU registers are a critical but limited resource, and directly reduce the runtime of a program by minimizing memory accesses. Over time, sizes of register files (RFs) have grown significantly in high performance and application-specific processors to handle both user-generated and compiler-generated memory references. However, large register files come at a noticeable area and power cost since they are designed with fast and leaky standard cells. This means analyzing RF usage in application code is important to achieve a high degree of utilization, energy efficiency, and a balance with other performance objectives.

Some CPU architectures have dedicated RFs for specific purposes to lower cycle count and power for specific types of instructions, and in order to not burden the general-purpose RFs that stores memory references (Mahlke et al. 1992; Espasa et al. 1995). Compilers often have capabilities to take advantage of special-purpose RFs (if any) in processor architectures to improve utilization of all the available RF resources including by generating memory references to minimize register spills and fills.

Open-Source Simulators

Over decades, the computer architecture community has developed and promoted various functional simulators for industry standard ISAs – MIPS, x86, Arm, RISC-V, etc. – for education, academic research, and sometimes for industry use too. To name a few, some of the well-known open-source simulators in use today are SimpleScalar (Burger and Austin 1997), Gem5 (Binkert et al. 2011, Abudaqa et al. 2018, Lowe-Power et al. 2020), QEMU (Qemu 2020), ESESC, and Spike (https://github.com/riscv/riscv-isa-sim). Some of these simulators support different ISAs, offer varying levels of configurability between supported CPU features, and extensibility for custom instructions.

Commercial CPU vendors provide accompanying proprietary software toolchains, which typically include functional ISA simulators often integrated with an integrated development environment (IDE), for easier evaluation and use of their CPUs. For example, Cadence Tensilica offers TurboXim – a fast, functional simulator for the Xtensa ISA – that can be used to quickly simulate target applications on Xtensa ISA, debug functional issues in the application, and profile the application to gather micro-architecture agnostic performance metrics as a first-order estimate of performance.

Cycle-Level Simulation

Definition

A more accurate measure of code performance on a target processor requires a cycle-level simulation of that processor. While it is not strictly necessary for a cycle-level simulator to also fully model the functionality of a processor or a feature, they often do, which also helps with more fine-grained functional debugging. Cycle-level simulators are more accurate as they model micro-architectural details of processor components. At the initial stages, standalone cycle-level models of individual processor components may be adequate to estimate performance of those components, but to achieve very high overall model accuracy, it may become necessary to model interactions between those components at a cycle level, such as interactions between the core pipeline, multiple branch predictors, and instruction cache to accurately model a complex event.

Performance Analysis

Cycle-level simulators primarily track the number of cycles needed to execute an instruction stream. They count pipeline stalls and replays, memory delays and conflicts, branch penalties among other multi-cycle events, which results in more accurate performance measurement at the cost of longer simulation time. A useful optimization for long simulation time is the use of statistical sampling. A hybrid simulation mode where an application is run in the cycle-level mode for a small percentage of time and run in the fast-functional mode for most of the time can yield a good balance between cycle-count accuracy and execution runtime.

Metrics and System Partitioning

Building a full-CPU cycle-level simulator is often a time-consuming and expensive endeavor. It is therefore a common practice initially to partition a system into components that can be modelled, verified, and analyzed reasonably independently. For instance, modelling resource and data hazards and interlocks allows the core pipeline to be analyzed separately from a standalone model for say the branch predictor. As the standalone simulators mature, the subtle interactions between them may be modelled. In our example, the precise cycles in the pipeline where various branch types are resolved for direction and target address become relevant in computing the misprediction penalties, and thereby the overall execution cycles.

Likewise, the memory subsystem of a processor can often be complex and is a good candidate for standalone cycle-level models before being integrated into the overall processor model. Instruction and data caches, tightly coupled (local) memories along with any banking capabilities, translation lookaside buffers (TLBs), and instruction and data prefetch blocks are all examples of components that can be modelled in isolation first. These independent models can be extended to model higher levels of memory (L-2, L-3 cache, system memory, etc.) to quickly analyze latency and throughput of bus transactions to identify potential performance bottlenecks even before any other parts of the processor are modelled and analyzed.

A popular environment that is used to develop full-system cycle-level models is SystemC, which is a C++ class library that extends standard C/C++ development environments to primarily facilitate hardware modelling.

Optimization

With the aid of cycle-level models – standalone or integrated – it is feasible to analyze application code more closely for performance bottlenecks. A basic analysis may just be determining upper/lower bounds of code performance for a given component or a processor configuration. The next stage of analysis could be either changing the code itself to measure performance on a given configuration, or changing the configuration to measure performance on a fixed piece of code (such as a benchmark). Sometimes, depending on the available feature set of a processor and the nature of the software algorithm, one may undertake data type precision analysis to optimize the algorithm to achieve highest performance on the target processor. It is often useful to also study how data placement in memory affects processor performance and if the target processor allows for it, take advantage of features like memory banking that allow concurrent memory accesses in the same cycle.

Configurability

An important characteristic of cycle-level models and simulators is the ability to configure various attributes of processor components. Configurability enables rapid design space exploration, model verification, and easy model extension to support new features, which may be something as trivial as adding a new instruction to the ISA. It is always a trade-off in simulator development on just how much generality to offer versus modelling fixed configurations with a high level of accuracy.

Open-Source Simulators

There are several open-source cycle-level CPU instruction set simulators (listed in "Open-Source Simulators") used by students, researchers, and engineers. These simulators often support multiple target architectures (such as Arm, SPARC, MIPS, Intel PC, PowerPC, RISC-V, etc.), multiple CPU devices, and generally some degree of configurability in the types of components being modelled. The nature of the CPU configuration being modelled would largely determine the accuracy and speed of any such simulator.

In addition, commercial CPU vendors often offer proprietary simulators for cycle-level simulation of their processors. As an example, the Cadence Tensilica Xtensa and Synopsys ARC processors come with proprietary simulators that support the full degree of configurability offered by their platform to allow SoC architects and software developers to analyze and optimize application code to the target processor.

A further approach to cycle-level simulation is to use RTL or translated RTL (Verilator). Arm provides Cycle Models of their CPU, which are generated based on the IP configuration by their IP Exchange portal (ARM 2021).

Hardware Emulation

Definition

Hardware emulation is the process of modelling a certain hardware component (such as a processor) purely in software or with different hardware. While simulators model the functions of a piece of hardware, emulators, on the other hand, attempt to also model the internal workings of that piece of hardware. Often, emulating new hardware designs with existing hardware offers many advantages, including faster program execution, more accurate functional verification, and in some cases, even PPA estimation. Thus, hardware emulation may encompass software models such as Qemu or hardware platforms such as Cadence Palladium, Synopsys Zebu, and Mentor Veloce. A good discussion of hardware-assisted emulation platforms can be found in Schirrmeister (2016).

Emulation Modes

There are various types of emulators or emulation modes based on the fidelity with which they emulate a system. Application mode models just enough components (CPU, memory, IO) to execute a workload. This model is often enough to measure performance but does not simulate any OS. Full platform mode models a broader set of components including network, disk, and other peripherals. Hybrid models use software emulation components in conjunction with hardware models (FPGA). In general, hardware emulators are meant to provide a high accuracy, stable, flexible, and fast model to execute real application software on a piece of target hardware that is being modelled on a different hardware.

Using Processor Simulators in System Modelling

In ▶ Chap. 26, "Methodologies for Design Space Exploration" by Andy Pimentel of this handbook, the author discusses use of processor simulators at various levels of abstraction in the design space exploration process, including the simulation of processor (really, processing) components and communication components at the RTL, cycle-accurate, and transaction-level abstractions. The execution-type processor simulators (ISS) discussed here fit at the cycle-accurate and transaction-level abstractions into system models. These may be proprietary, open-source (e.g., SystemC), or third-party commercial system modelling environments. We illustrate these concepts using a particular family of processor ISS models (the cycle-accurate and transaction-level models provided with the Cadence Xtensa processors), but this is merely for illustrative purposes. Many other processor simulators from both open-source and commercial providers have similar characteristics. ▶ Chapter 27, "Virtual Prototyping of Processor-Based Platforms" by Tim Kogel provides considerable extra detail about proprietary third-party system modelling environments.

The Cadence Xtensa configurable processor IP originally included a proprietary C-based modelling API and environment known as XTMP (Martin et al. 2010). The

processor ISS, as discussed earlier, was generated to support the exact configuration and extended ISA of the particular processor. To an almost complete extent, the XTMP modelling approach was subsumed by a SystemC approach, XTSC, that allowed much easier integration into open-source and third-party commercial system modelling tools, so we will discuss only the SystemC-based approach here.

The processor models have many introspective model-query methods that permit all the relevant characteristics of a particular configuration (for example, numbers and types of memory interfaces) to be determined, which allows the dynamic creation of integration wrappers for the processor models.

In addition, the interfaces supported include cycle-accurate transaction-level interfaces, fast-functional transaction-level interfaces, and cycle-accurate pin-level interfaces, which allow integration into several different types of system models.

The XTSC modelling environment also includes several generic device models (e.g., memories, connectors, routers, DMA) that are configurable and allow the manual construction of relatively complex system models for design space exploration.

XTSC supports several system simulation use models: single-processor simulation with models of various external devices, including memories and peripheral blocks; multiprocessor simulation with both Xtensa and other processors, complex buses or networks-on-chip, and hardware accelerators; mixed system and HDL simulation using pin-level transactors and interfaces to Verilog simulation of hardware blocks; and virtual prototype simulation. All these use cases permit, where supported, both cycle-accurate simulation and fast-functional (compiled code) simulation.

The system modelling tools then support software development and functional verification; system and software profiling; and debugging.

When used as a standalone SystemC-based modelling tool, XTSC supports many utility functions that otherwise are offered in third-party environments, such as logging, easy setup and integration, out-of order simulation in fast-functional mode, and direct memory access methods. Although developed prior to SystemC TLM2 abstractions, it supports TLM2 using transaction adaptors.

The use of an industry standard such as SystemC allows easier integration of the processor simulators into third party proprietary ESL system tools. Over the years, such integrations have been done with CoWare, Virtio and VaST System Technology (all acquired by Synopsys, and current integration with Synopsys virtual prototyping tools Platform Architect Ultra and Virtualizer), Rockwell Semiconductor Maxsim (that became a spinoff Axys Design Automation, then part of Arm, then part of Carbon Design, and finally part of Arm again), Imperas (OVPSim), Virtutech Simics (then part of Intel's Wind River and then Wind River spun back out of Intel), and Mirabilis. Some of these integrations are historical and not current, but they show that use of SystemC XTSC instead of the earlier proprietary XTMP made integration easier.

As noted earlier, Qemu (Bellard 2005, Qemu 2020) has been used for many years to build instruction set simulator models for many different processor families – including x86, Sparc, PowerPC, MIPS, Xtensa, and notably and recently, RISC-

V. It has then been integrated into many system or platform models (Lonardi and Pravadelli 2014; Qemu System 2020) where models of peripherals and buses can be combined with single or multiple processor models (possibly heterogeneous) to produce system virtual prototype models that can be used as in the earlier discussion for software development, functional verification, profiling, and debugging.

Some of the full chip platforms that have been built, as well as using the processors cited earlier, include various Arm boards, Coldfire, PC emulators, IBM mainframe emulators, and many more.

Summary Table Comparing Various CPU Modelling Abstractions

Type	Abstraction level	Speed	Best for	Also used for
Functional simulation	Architectural – instruction level	<500M instructions per second	Functional verification, SW development	Gross performance estimation
Cycle-level simulation	Micro-architectural – clock precise	~10k–10M instructions per second	Performance characterization	Gross PPA estimation
Hardware emulation (FPGA and custom processor)	Full system including peripherals	>1M instructions per second	Functional and performance validation	Accurate PPA measurement

Examples

The methodologies discussed in this chapter have been applied over the years to many different processors with ISAs drawn from several sources. Notable among them are Synopsys ARC cores, Cadence Tensilica CPUs and DSPs, Ceva processors, and RISC-V processors including academic research projects and commercial offerings from providers such as Greenwaves, SiFive, Andes, Codasip, and Syntacore. All these offerings allow some measure of ISA customization to allow the cores to be tuned to particular application spaces, whether done by the provider or the end user. In addition, standard offerings from Arm and the Intel x86 world give fixed ISA alternatives to the ability to customize an application-specific instruction set processor (ASIP) (Ienne and Leupers 2007).

Applications targeted with these methodologies include wireless communications (Rowen et al. 2009, Rowen 2012, Heine 2016], vision processing (Rowen 2015), imaging and AI (Efland et al. 2016), cryptography (Marshall et al. 2020), cryptography in vehicle-to-vehicle communications (Ogawa et al. 2019), and advanced 64-bit math in a 32-bit RISC (Bit 2019).

This rich set of applications and the need to choose and possibly tune the ISA reinforces the importance of the methodologies illustrated here. To illustrate more deeply, we will summarize the design methodology used in Ogawa (2019). The authors developed a design to better support cryptographic applications in vehicle-to-vehicle communications. They went through the following steps:

1. They carried out a thorough analysis of the security issues in vehicle-to-vehicle communications and the cryptographic solutions available. This led them to focus on accelerating the PRESENT block cipher, and Galois Field multiplication (for Curve25519 arithmetic functions), as extensively discussed in the paper.
2. They considered two basic approaches to accelerating the cipher encryption, decryption and key update round, and Galois Field multiplication: adding independent, memory-mapped, coprocessor modules to a main CPU, or using an extensible, configurable processor and adding new instructions to it, as well as making configuration choices to better suit the algorithm requirements.
3. Considerations of processing latency led them to the extensible processor solution. In addition, they analyzed the algorithm and decided that both instruction extensions and choosing dual-data tightly coupled memory banks (X-Y memories) would give them the desired algorithm performance with low logic overhead and reduced energy consumption.
4. Although they acknowledged several possible choices existed for a configurable, extensible processor platform with which to continue, they chose the Synopsys DesignWare ARC EM processor family as the basis for their work. It had the basic features of instruction extensibility and dual X-Y memory options that were important.
5. Using their analysis of the PRESENT algorithms and requirements, they defined additional registers and candidate instruction extensions to allow speedup of the block encryption, key schedule, and block decryption algorithms. The cypher block size was 64 bits and hence 64-bit registers were added to support both 64-bit and 128-bit datapaths.
6. The design of the instruction extensions used the Synopsys APEX methodology to implement the operations, the new registers, and their physical implementation and linkage to the ARC processor.
7. In addition, they designed additional instruction extensions for Galois Field multiplication as required by the target application. This used a Synopsys execution profiling tool running against the code compiled to the ARC processor, which makes use of an appropriate ISS model of the processor.
8. The design of these additional extensions involved trade-offs in the size of the accumulator required for 256-bit by 256-bit full multiplication, and they designed a 288-bit accumulator and shifter to optimize the area and latency to the algorithm (rather than using a full 512-bit accumulator). They also shared resources for the various operations involved in the multiplier operation set.
9. They also worked on optimizing the code and the layout of data to make optimal use of the dual X-Y tightly coupled memories for both the block encryption/de-

cryption and key schedule operations, and the Galois Field multiplication. This was fairly intricate work; the paper has appendices which provide details on the operations.

10. To test the revised benchmark code (incorporating the instruction extensions and the data layout directives), the authors used the Synopsys cycle-accurate processor model. They used test vectors from various industry standards sites for the cryptographic algorithms. PPA was estimated by targeting an FPGA implementation of the resulting processor, although of course it could have been implemented in an SoC. Details are given in the paper of the generated DesignWare ARC xCAM cycle-accurate models, incorporating the APEX instruction extensions.
11. The final measurements indicated that application of both instruction extensions and X-Y memory sped up the PRESENT block cipher application by 17–34 times, at a cost of 4% in FPGA LUTS and 8% in registers.
12. Similarly, for Galois Field Multiplication, the Curve25519 algorithm sped up by 2.5 times at a relatively low cost of 9% in FPGA LUTS, 15% in registers, and a small number of DSP blocks and Carry8 primitives.
13. In their conclusion, they discuss that other extensible and configurable processors could have been the targets for this work, and the methodology approach could be used with other cryptographic algorithms.

To summarize, the work in Ogawa (2019) is a very good illustration of the design and methodology approaches discussed in this chapter in practical use.

Conclusion

In this chapter, we have discussed two orthogonal but related concepts: how to simulate instruction set processors; and how to characterize them, using processor simulators, in order to make choices as to what processor(s) to use for specific design projects, and whether they will be standard off-the shelf designs or will be customized using a number of different possible approaches.

Characterizing processors can be done with standard benchmarks, or application-specific design code, or a suitable mixture of both. We have reviewed several the standard benchmarks used over the years and have given examples of how design code may be used.

Finally, we have shown via examples of processor characterization how these concepts can be applied to several interesting application areas.

References

Abudaqa AA, Al-Kharoubi TM, Mudawar MF, Kobilica A. Simulation of ARM and x86 microprocessors using in-order and out-of-order CPU models with Gem5 simulator. In: 2018 5th international conference on Electrical and Electronic Engineering (ICEEE), May 2018. IEEE, Istanbul, Turkey, pp 317–322

Arm IP. Exchange portal web page (2021). https://ipx.arm.com
BDTi Inc (2020). https://www.bdti.com. Accessed 5 Oct 2020
Bellard F (2005) Qemu, a fast and portable dynamic translator. In: Proceedings of the USENIX annual technical conference, Anaheim, 2005. pp 41–46
Binkert N, Beckmann B, Black G, Reinhardt SK, Saidi A, Basu A, Hestness J, Hower DR, Krishna T, Sardashti S, Sen R, Sewell K, Shoaib M, Vaish N, Hill MD, Wood DA (2011) The gem5 Simulator. In: SIGARCH Computer Architecture News, 39, 2 (May 2011), 1–7. https://doi.org/10.1145/2024716.2024718
Bit A (2019) 64-bit Custom Math ISA in Configurable 32-bit RISC Processor, In: Kumar A, Paprzycki M, Gunjan V (eds) Proceedings of the first international conference on data science, machine learning and applications, (ICDSMLA 2019), Springer, Singapore
Brown M, Jenevein RM, Ullah N (1998) Memory access pattern analysis. In: Proceedings of the workload characterization: methodology and case studies (WWC '98). IEEE Computer Society, USA, 105
Burger D, Austin TM (1997) The SimpleScalar tool set, version 2.0. In: ACM SIGARCH Computer Architecture News 25(3):13–25
Codasip Studio web page (2021). https://codasip.com/products/codasip-studio/
Github Coremark benchmark from EEMBC. https://github.com/eembc/coremark. Accessed 6 Oct 2020
Curnow HJ, Wichmann BA (1976) A synthetic benchmark. Comput J 19(1):43–49
Dongara JJ, Luszczek P, Petitet A (2003) The LINPACK Benchmark: past, present and future. Concurrency and Computation: Practice and Experience 15(9):803–820
EEMBC (2020) Embedded microprocessor benchmark consortium (2020). https://www.eembc.org. Accessed 5 Oct 2020
EEMBC Coremark (2020) Embedded Microprocessor Benchmark Consortium (2020). https://www.eembc.org/coremark/. Accessed 5 Oct 2020
Efland G et al (2016) High performance DSP for Vision, Imaging and Neural Networks, Hot Chips, Cupertino, 2016
Embench™: A modern embedded benchmark suite (2020). http://embench.org. Accessed 5 Oct 2020
Espasa R, Valero M, Padua D, Jiminez M, Ayguade E (1995) Quantitative analysis of vector code. In: Proceedings Euromicro workshop on parallel and distributed processing, San Remo, Italy, 1995, pp. 452–461. https://doi.org/10.1109/EMPDP.1995.389176
Heine D (2016) Multi-Purpose, low-power DSPs for mobile and other markets, Linley Mobile and Wearables Conference, Santa Clara, 2016
Ienne P, Leupers R (eds) (2007) Customizable embedded processors. Morgan-Kaufmann Publishers, Burlington, MA
Leibson S (2006) Using performance metrics to select microprocessor cores for IC design. In: Scheffer L, Lavagno L, Martin G (eds) The EDA handbook, volume I, chapter 10, 1st edn. CRC Press/Taylor and Francis, Boca Raton
Leibson S (2016) Using performance metrics to select microprocessor cores for IC design. In: Scheffer L, Lavagno L, Markov I, Martin G (eds) The EDA handbook, vol I, chapter 10, 2nd edn. CRC Press/Taylor and Francis, Boca Raton
Lonardi A, Pravadelli G (2014) On the Co-simulation of SystemC with QEMU and OVP virtual platforms. In: 22th IFIP/IEEE International Conference on Very Large Scale Integration – system on a chip (VLSI-SoC 2014), Playa del Carmen, Mexico, October, 2014. 110–128
Lowe-Power J et al (2020) The gem5 Simulator: version 20.0+. https://arxiv.org/pdf/2007.03152.pdf. 30 Sept 2020
Mahlke SA, Chen WY, Chang PP, Hwu WW (1992) Scalar program performance on multiple-instruction-issue processors with a limited number of registers. In: Proceedings of the twenty-fifth Hawaii international conference on system sciences, Kauai, HI, USA, 1992, pp 34–44, vol 1. https://doi.org/10.1109/HICSS.1992.183141
Marsh J (2020) ARM helium technology: M-Profile Vector Extension (MVE) for ARM Cortex-M Processors, arm Education Media, Cambridge

Marshall B, Newell GR, Page D, Saarinen M, Wolf C (2020) The design of scalar AES instruction set extensions for RISC-V. In: IACR transactions on cryptographic hardware and embedded systems (TCHES). 2020

Martin G, Nedeljkovic N, Heine D (2010) Configurable, extensible processor system simulation. In: Leupers R, Temam O (eds) Processor and System-on-Chip simulation. Springer, Heidelberg, pp 293–308

The Mathworks (2020). http://www.mathworks.com. Accessed 5 Oct 2020

McCallum JC, Chua T (1987) A synthetic instruction mix for evaluating microprocessor performance. IEEE Micro, May/June, 1987. pp 63–80

Mediatek (2020). https://www.mediatek.com/products/smartphones. Accessed 5 Oct 2020

Misra S, Alfa AA, Olaniyi MO, Adewale SO (2014) Exploratory study of techniques for exploiting instruction-level parallelism. 2014 Global Summit on Computer & Information Technology (GSCIT), Sousse, 2014, pp 1–6. https://doi.org/10.1109/GSCIT.2014.6970103

Ogawa H, Luther T, Ricardini J, Cunha H, Simplicio Jr. M, Aranha D, Derwig R, Patil H (2019) Accelerated V2X provisioning with extensible processor platform. IACR Cryptol. ePrint Arch. 2019: 1039 (2019)

Qualcomm (2020). https://www.qualcomm.com/products/application-processors. Accessed 5 Oct 2020

Patterson D (2020) Embench™: recruiting for the long overdue and deserved demise of Dhrystone as a benchmark for embedded computing. Computer Architecture Today. https://www.sigarch.org/embench-recruiting-for-the-long-overdue-and-deserved-demise-of-dhrystone-as-a-benchmark-for-embedded-computing/. Accessed 5 Oct 2020

Qemu. www.qemu.org. Accessed 5 Oct 2020

Qemu System Emulator Targets (2020). https://www.qemu.org/docs/master/system/targets.html. Accessed 5 Oct 2020.

Risc-V International (2020). http://www.riscv.org. Accessed 5 Oct 2020

Risc-V International Exchange Cores and SOCs (2020). https://www.riscv.org/exchange/cores-socs/. Accessed 5 Oct 2020

Rowen C et al (2009) A DSP architecture optimised for wireless baseband. In: International symposium on System-on-Chip. Tampere, Finland, 2009

Rowen C (2012) Power/performance breakthrough for lte advanced handsets, Linley Mobile Conference, Santa Clara, April 16, 2012

Rowen C (2015) Instruction set innovation in fourth generation vision DSPs, Linley Processor Conference, Santa Clara, 2015

Schirrmeister F, Bershteyn M, Turner R (2016) Hardware-assisted verification and software development. In: Scheffer L, Lavagno L, Markov I, Martin G (eds) The EDA handbook, Volume I, chapter 19, 2nd edn. CRC Press/Taylor and Francis, Boca Raton

Sherwood T, Perelman E, Hamerly G, Calder B (2002) Automatically characterizing large scale program behavior, ASPLOS X: Proceedings of the 10th international conference on architectural support for programming languages and operating systems, San Jose, October 2002. pp 45–57. https://doi.org/10.1145/605397.605403

Standard Performance Evaluation Corporation [SPEC CPU] (2020). https://www.spec.org/benchmarks.html. Accessed 5 Oct 2020

Sudarsanam A, Malik S (1995) Memory bank and register allocation in software synthesis for ASIPs. In: Proceedings of IEEE international conference on Computer Aided Design (ICCAD), San Jose, CA, USA, 1995, pp. 388–392. https://doi.org/10.1109/ICCAD.1995.480145

Synopsys ASIP Designer Web page (2021). https://www.synopsys.com/dw/ipdir.php?ds=asip-designer

Weiss A (2002) Dhrystone benchmark: history, Analysis, "Scores" and Recommendations: White paper. Embedded Microprocessor Benchmark Consortium (EEMBC). https://www.eembc.org/techlit/datasheets/dhrystone_wp.pdf. Accessed 5 Oct 2020

Wolberg G (1990) Digital image warping. Wiley-IEEE Computer Society Press, Los Alamitos

Methodologies for Design Space Exploration

26

Andy D. Pimentel

Contents

Introduction . . . 916
DSE: The Basic Concepts . . . 917
Two Basic Ingredients of DSE . . . 919
Y-Chart-Based DSE . . . 920
Evaluation of a Single Design Point . . . 922
Simulative Fitness Evaluation . . . 922
Analytical Fitness Evaluation . . . 926
Searching the Design Space . . . 927
GA-Based DSE . . . 928
Optimizing GA-Based DSE . . . 931
Multi-application Workload Models . . . 932
Scenario-Based DSE . . . 933
Application Exploration . . . 937
NAS by Means of Evolutionary Piecemeal Training (EPT) . . . 937
Conclusion and Outlook . . . 940
References . . . 942

Abstract

In this chapter, an overview of techniques and methods for the design space exploration (DSE) of embedded systems is presented. DSE is the critical design process in which system designs are modeled, evaluated, and, eventually, optimized for the various extra-functional system behaviors, such as performance, power or energy consumption, and cost. The discussion is organized along the lines of the two primary elements of DSE, namely, the evaluation of single design points and the search strategy for covering the design space.

A. D. Pimentel (✉)
Parallel Computing Systems Group, University of Amsterdam, Amsterdam, The Netherlands
e-mail: a.d.pimentel@uva.nl

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_23

Keywords

Design space exploration · Multi-objective optimization · Performance modeling · Genetic algorithms

Introduction

Designers of modern embedded computer systems face several daunting challenges since these systems typically have to meet a range of stringent and often conflicting, design requirements. As many embedded systems target mass production and battery-based devices or devices that cannot use active cooling, they should be cheap and power efficient. Mission- and safety-critical embedded computer systems, like those in the avionics and space domains, usually also demand high levels of dependability, which is becoming even more important as the levels of system autonomy rise. Moreover, a great deal of these systems must, increasingly, support multiple applications and standards for which they often need to provide real-time performance. For example, mobile devices must support a variety of different standards for communication and coding of digital contents. In addition, many of these systems also need to provide a high degree of flexibility, allowing them to be easily updated and extended with future applications and standards. This calls for a high degree of programmability of these systems, whereas performance, power consumption, and cost constraints require implementing substantial parts of these systems in dedicated hardware blocks. As a result, modern embedded systems often have a *heterogeneous* multiprocessor system architecture. They consist of processors that range from fully programmable cores to fully dedicated hardware blocks for time-critical application tasks. Increasingly, the components in such systems are integrated onto a single chip, yielding heterogeneous multiprocessor system-on-chip (MPSoC) architectures (Wolf et al. 2008).

To cope with the design complexity of such systems, a new design methodology has emerged in the past 15 to 20 years, called system-level design (see chapter "Electronic System-Level Design"). It aims at raising the level of abstraction of the design process to improve the design productivity. Key enablers to this end are the use of MPSoC platform architectures to facilitate reuse of IP components and the concept of high-level system modeling and simulation (Keutzer et al. 2000; Sangiovanni-Vincentelli and Martin 2001). The latter allows for capturing the behavior of platform components and their interactions at a high level of abstraction. As such, these high-level models minimize the modeling effort and are optimized for execution speed and can therefore be applied during the very early design stages to perform *design space exploration* (DSE) (Gries 2004; Pimentel 2017). During such DSE, a large variety of different design alternatives can be explored, such as the number and type of processors deployed in the platform architecture, the type of interconnection network used to connect system components, or the spatial binding and temporal binding (i.e., scheduling) of application tasks to processor cores. It is of paramount importance to start performing such DSE as early as possible in

the design process because the considered design choices may heavily influence the success or failure of the final product. However, the process of DSE also is highly challenging since the design space that needs to be explored typically is vast, especially during the early stages of design. For instance, the design space for exploring different mappings of application tasks to processing resources – and trying to optimize the mapping for, e.g., system performance or power consumption – exponentially grows with the number of application tasks and processors in the system and is known to be an NP-hard problem (Singh et al. 2013). Therefore, the development of efficient and effective DSE methods has received significant research attention in recent years. In this chapter, an overview will be provided of the various aspects involved in DSE of embedded systems.

DSE: The Basic Concepts

During the DSE of embedded systems, multiple optimization *objectives* – such as performance, power/energy consumption, and cost – should be considered simultaneously. This is called multi-objective DSE (Pimentel 2017). Since the objectives are often in conflict, there cannot be a single optimal solution that simultaneously optimizes all objectives. Therefore, optimal decisions need to be taken in the presence of trade-offs between design criteria.

Given a set of m decision variables, which are the degrees of freedom (e.g., MPSoC system parameters like the number and type of processors, application mapping, etc.) that are explored during DSE, a so-called fitness function must optimize the n objective values. The fitness function is defined as:

$$f_i : R^m \rightarrow R^1 \tag{1}$$

A potential solution $x \in R^m$ is an assignment of the m decision variables. The fitness function f_i translates a point in the solution space X into the ith objective value (where $1 \leq i \leq n$). For example, a particular fitness function f_i could assess the performance or energy efficiency of a certain solution x (representing a specific design instance). As illustrated in Fig. 1, the combined fitness function $f(x)$ subsequently translates a point in the solution space into the objective space Y. Formally, a multi-objective optimization problem (MOP) that tries to identify a solution x for the m decision variables that minimizes the n objective values using objective functions f_i with $1 \leq i \leq n$:

$$\begin{aligned} \text{minimizes } y &= f(x) = (f_1(x), f_2(x), \ldots, f_n(x)) \\ \text{where } x &= (x_1, x_2, \ldots, x_m) \in X \\ y &= (y_1, y_2, \ldots, y_n) \in Y \end{aligned}$$

Here, the decision variables x_i (with $1 \leq i \leq m$) usually are constrained. These constraints make sure that the decision variables refer to valid system configurations

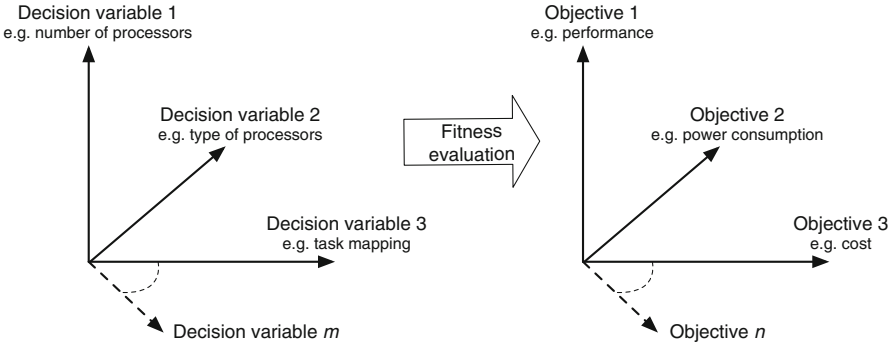


Fig. 1 The design space broken down in solution and objective space

(e.g., using not more than the available number of processors, using a valid mapping of application tasks to processing resources, etc.), i.e., x_i are part of the so-called feasible set. In the remainder of this discussion, a minimization procedure is assumed, but without loss of generality, this minimization procedure can be converted into a maximization problem by multiplying the fitness values y_i with -1.

With an optimization of a single objective, the comparison of solutions is trivial. A better fitness (i.e., objective value) means a better solution. With multiple objectives, however, the comparison becomes nontrivial. Take, for example, two different MPSoC designs: a high-performance MPSoC and a slower but much cheaper one. In case there is no preference defined with respect to the objectives and there are also no restrictions for the objectives, one cannot say if the high-performance MPSoC is better or the low-cost MPSoC. A MOP can have even more different objectives, like the performance, energy consumption, cost, and reliability of an MPSoC-based embedded system. To compare different solutions in the case of multiple objectives, the Pareto dominance relation is generally used. Here, a solution $x_a \in X$ is said to dominate solution $x_b \in X$ if and only if $x_a \prec x_b$:

$$x_a \prec x_b \iff \forall i \in \{1, 2, \ldots, n\} : f_i(x_a) \leq f_i(x_b) \wedge \exists i \in \{1, 2, \ldots, n\} : f_i(x_a) < f_i(x_b)$$

Hence, a solution x_a dominates x_b if its objective values are superior to the objective values of x_b. For all of the objectives, x_a must not have a worse objective value than solution x_b. Additionally, there must be at least one objective in which solution x_a is better (otherwise they are equal).

An example of the dominance relation is given in Fig. 2, which illustrates a two-dimensional MOP. For solution H, the dominance relations are shown. Solution H is dominated by solutions B, C, and D as all of them have a lower value for both f_1 and f_2. On the other hand, solution H is superior to solutions M, N, and O. Finally,

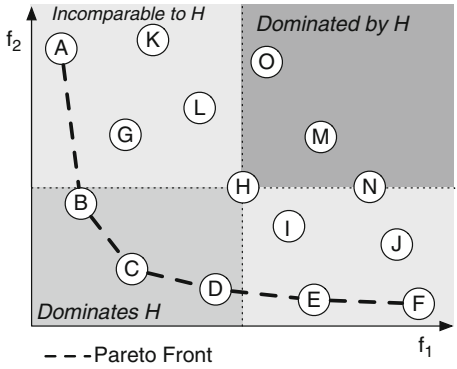


Fig. 2 A Pareto front and an example of the dominance relation

some of the solutions are not comparable to H. These solutions are better for one objective but worse for another.

The Pareto dominance relation only provides a partial ordering. For example, the solutions A to F of the example in Fig. 2 cannot be ordered using the ordering relation. Since not all solutions $x \in X$ can be ordered, the result of a MOP is not a single solution but a front of non-dominated solutions, called the *Pareto front*. A set X' is defined to be a Pareto front of the set of solutions X as follows:

$$\{x \in X' \mid \nexists x_a \in X : x_a < x\}$$

The Pareto front of Fig. 2 contains six solutions: $A - F$. Each of these solutions does not dominate the other. An improvement on objective f_1 is matched by a worse value for f_2. Generally, it is up to the designer to decide which of the solutions provide the best trade-off.

Two Basic Ingredients of DSE

The search for Pareto optimal design points with respect to multiple design criteria as targeted by DSE entails two distinct elements (Gries 2004; Pimentel 2017):

1. The evaluation of a single design point using the fitness function(s) regarding all the objectives in question like system performance, power/energy consumption, and so on
2. The search strategy for covering and navigating through the design space, spanned by the decision variables x_i (with $1 \leq i \leq m$), during the DSE process

Figure 3 shows a simple taxonomy for DSE approaches, recognizing the above two DSE elements as well as different properties of these DSE elements. Please note that these properties typically cannot be considered in pure isolation as they can be interdependent and even conflicting with each other. As will be discussed in more

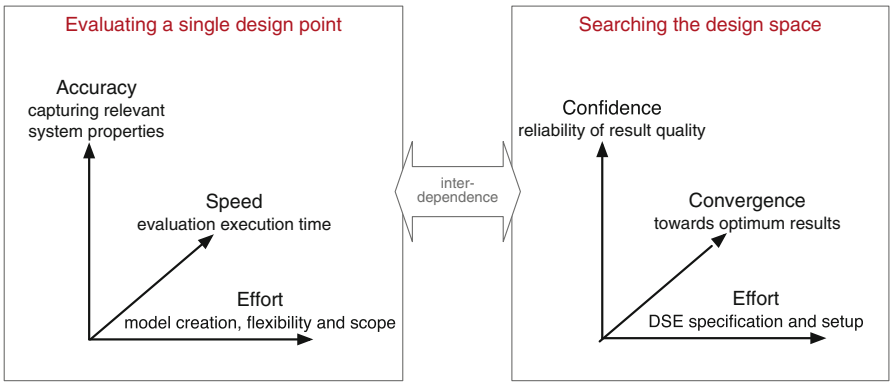


Fig. 3 A taxonomy for DSE approaches. (Taken from Thompson 2012)

detail later on, there usually exists a trade-off between the accuracy and speed with which the fitness of single design points can be evaluated. In addition to this, the various fitness evaluation techniques also differ with respect to the implementation effort and the capability of evaluating the fitness for a wide range of systems, involving issues such as modularity, reusability of models, etc.

Regarding the search strategy aspect of DSE, the confidence property denotes the degree of certainty that the design points returned by the DSE include the true optimum or, alternatively, how close they are to the true optimum. In many search algorithms, confidence is obtained by avoiding local optima and ensuring sufficient design space coverage. Clearly, an exhaustive search in which every single point in the design space is evaluated and compared would provide a 100% confidence. However, such exhaustive search is usually prohibited due to the sheer size of the design space. In those cases, as will be discussed later on, search techniques based on metaheuristics can be used to search the design space for optimal solutions using only a finite number of design point evaluations. The convergence property denotes the speed of evaluating a range of design points and, more specifically, the rate at which the DSE search algorithm manages to converge to an optimum. Finally, analogous with the effort property in the case of evaluating a single design point, the effort for searching the design space refers to the implementation of the search method and setting its parameters, as well as setting up, running, and evaluating the results of the exploration experiment.

Y-Chart-Based DSE

Many system-level fitness evaluation and DSE methods and tools in the embedded systems domain are based on the Y-chart methodology (Kienhuis et al. 2002; Balarin et al. 1997), which is illustrated in Fig. 4. This implies that these DSE methods separate application models (or workload models) and architecture models

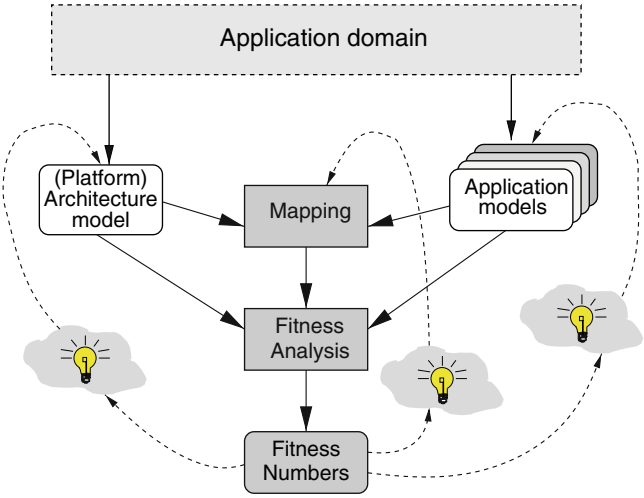


Fig. 4 Y-chart-based DSE (Kienhuis et al. 2002; Balarin et al. 1997)

while also recognizing an explicit mapping step to map application tasks onto architecture resources (i.e., bind tasks to processing elements in space and time). In this approach, an application model – derived from a specific application domain – describes the functional behavior of an application workload in a timing and architecture independent manner. An MPSoC (platform) architecture model – which usually has been defined with the application domain in mind – defines architecture resources and captures their extra-functional behavior, i.e., behavior in terms of performance, power consumption, cost, etc. To perform quantitative analysis of the fitness of a design point, application models are mapped onto the architecture model under investigation, after which the fitness of each application-architecture combination can be evaluated. Subsequently, the resulting fitness numbers may be used by the search algorithm of a DSE process to change the architecture, restructure/adapt the application(s), or modify the mapping of the application(s). These actions are illustrated by the light bulbs in Fig. 4.

Essential in this methodology is that an application model is independent from architectural specifics, assumptions on hardware/software partitioning, and timing characteristics. As a result, application models can be reused in the exploration cycle. For example, a single-application model can be used to exercise different hardware/software partitionings or can be mapped onto a range of architecture models, possibly representing different MPSoC architecture designs or modeling the same architecture design at various levels of abstraction. The latter refers to the gradual refinement of architecture models (e.g., Pimentel et al. 2006; Thompson et al. 2006). As design decisions are made, a designer typically wants to descend in abstraction level by disclosing more and more implementation details in an architecture model. Eventually, such refinement can bring an initially abstract

architecture model closer to the level of detail where it is possible to synthesize an implementation (Nikolov et al. 2008; Thompson et al. 2007; Stefanov et al. 2017).

In the next two sections, a more detailed overview will be provided of the different techniques, and their properties, applied in each of the two aforementioned elements of DSE, i.e., fitness evaluation of a single design point and searching the design space.

Evaluation of a Single Design Point

Methods for evaluating the fitness of a single design point in the design space roughly fall into one of three categories: (1) measurements on a (prototype) implementation, (2) simulation-based evaluations, and (3) estimations based on an analytical model. Each of these methods has different properties with regard to evaluation time and accuracy. Evaluation of prototype implementations provides the highest accuracy, but long development times prohibit evaluation of many design options. Analytical estimations are considered the fastest, but accuracy is limited since they are typically unable to sufficiently capture particular intricate system behavior. Simulation-based evaluation fills up the range in between these two extremes: both highly accurate (but slower) and fast (but less accurate) simulation techniques are available (see also ▸ Chap. 25, "Processor Simulation and Characterization"). This trade-off between accuracy and speed is very important, since successful DSE depends both on the ability to evaluate a single design point and being able to efficiently search the entire design space. As present DSE efforts in the domain of embedded systems design usually use simulation or analytical models to evaluate single design points, the remainder of this section will focus on these methods.

Simulative Fitness Evaluation

Simulating system components can be performed at different levels of abstraction. The higher the abstraction level, the less intricately the system components are modeled and, therefore, the higher the simulation speed is. Evidently, such efficiency improvements come at the cost of a less accurate fitness estimation because of the fact that particular system details are not taken into account. This simulation speed-accuracy trade-off is shown in Fig. 5. This figure depicts several widely used simulation abstraction levels, and it does so for both the simulation of processor components and the simulation of communication between system components.

For both the simulation of processor and communication components, the lowest level of abstraction for simulating a digital system is the register-transfer level (RTL). At this level of abstraction, the flow of digital signals between registers and combinational logic is explicitly simulated. This yields a highly accurate but also very slow simulation. As a result, the use of RTL simulation in the process of DSE is confined to only relatively small and narrow design spaces focusing on,

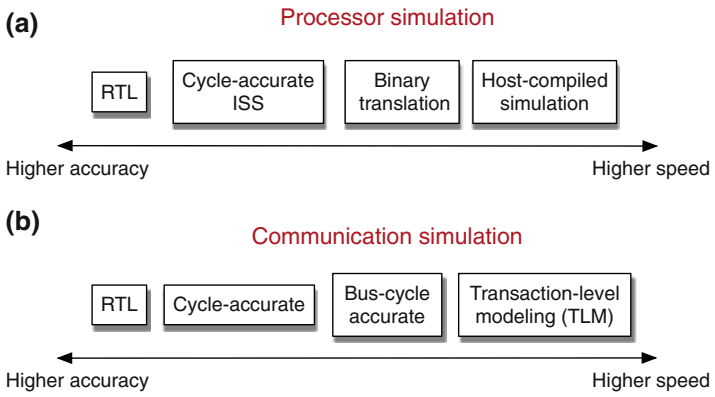


Fig. 5 Different levels of abstraction for (**a**) simulating processors and (**b**) simulating communication

for example, the design of one specific system component. Performing system-level DSE is infeasible using RTL simulation.

Raising the level of abstraction, one can simulate system components at the cycle-accurate level. This means that the system components are simulated on a cycle-by-cycle basis and, as such, that the simulated system state conforms to the cycle-by-cycle behavior of the target design. This results in more efficient simulation as compared to RTL simulation at the cost of a somewhat reduced accuracy since the system state in between cycles is not accounted for. Cycle-accurate simulation is a popular technique for simulating microprocessors (see also ▸ Chap. 25, "Processor Simulation and Characterization"): so-called cycle-accurate instruction set simulation (ISS). These ISS simulators try to capture the cycle-by-cycle behavior of the micro-architectural components of a microprocessor, such as the pipeline logic, out-of-order processing, branch predictors, caches, and so on. To account for power consumption behavior, ISS simulators often use activity-based power models that accumulate the power consumption of the relevant micro-architecture components based on their activity ratio. A good example is the widely used cycle-accurate Gem5 ISS (Binkert and et al.: 2011), which can be extended to also support area and power predictions using activity-based modeling frameworks such as CACTI (Thoziyoor et al. 2008) and McPAT (Li et al. 2013). Although these ISS simulators can be deployed to perform micro-architectural DSE for processor components, they are generally still too slow for performing full system-scale DSE of multicore-based embedded systems.

In cycle-accurate ISS simulators, the fetching, decoding, and execution of instructions are explicitly simulated. To further optimize the speed of such simulators, one could translate the instructions from the target binary to be simulated to an equivalent sequence of instructions (using static or dynamic just-in-time translation) that can be executed on the simulation host computer. This so-called *binary translation* technique, which is, for example, deployed in the widely used QEMU

simulator (Bellard 2005), aims at reducing the overhead of explicitly simulating the instruction fetch and decode stages. The translated instruction sequences are often instrumented with additional code to keep track of the extra-functional behavior such as timing and power consumption, of the original code as it would have been executed on the target processor. In some cases, however, ISS simulators and especially binary translation-based simulators only focus on mimicking the functional behavior and do not capture the extra-functional behavior of the target processor. In these cases, they are usually referred to as *emulators* rather than simulators.

For simulating communication between system components, one could use so-called bus-cycle-accurate simulation (Cai and Gajski 2003) to speed up the simulation process. In this type of simulation, all signals of the communication bus are modeled explicitly in a cycle-accurate fashion, but this accuracy is only maintained for the signals on the communication bus and not for the logic around it. The surrounding components can thus use more abstract timing models.

Raising the abstraction level even further for processor simulation yields so-called host-compiled simulation (Ceng et al. 2009; Bringmann et al. 2015). In this technique, the source code of the target program is directly compiled into a binary program that can run on the host computer. In addition, and similar to the binary translation technique, the source code can be instrumented with a timing and power consumption model based on the target architecture. Since this type of simulation is efficient as it directly executes target programs on the host computer, it is very suitable for system-level DSE. However, at this level of abstraction, it is difficult to accurately capture intricate micro-architectural behavior, like pipeline and caching behavior. Another drawback of this simulation approach is that one needs to have access to the source code of a target program.

For simulating communications, *transaction-level modeling (TLM)* (Cai and Gajski 2003) provides the highest level of abstraction. In TLM, communication details at the level of signals and protocols are abstracted away by means of encapsulation into entire transactions between system components. At this level, the emphasis is more on the functionality of the data transfers, i.e., what data are transferred to and from what locations, rather than on their actual implementation. Evidently, the extra-functional behavior in TLM simulation models is also captured at the level of entire transactions.

The above processor simulation techniques are all execution-driven simulation methods as they are directly driven by the execution of a program. Alternatively, there are also *trace-driven simulation* techniques in which the simulation is driven by event traces that have been collected through the execution of a program (e.g., Butko et al. 2015; Castrillon et al. 2010). These trace events can focus on the evaluation of specific system elements such as memory access address traces for cache simulation (Uhlig and Mudge 1997). However, an event trace may also consist of the full sequence of executed instructions, thereby allowing full, trace-driven microprocessor simulation for the purpose of performance and/or power estimation. To optimize for simulation speed, the trace events may also represent computations

(and, if needed, communications) at a higher level of abstraction than the level of machine instructions, like at the level of the execution of basic blocks or even entire functions. Another advantage of trace-driven simulation is the fact that the event traces often only need to be generated once (i.e., executing the program to collect the traces once), after which they can be reused in the DSE process. Drawbacks of trace-driven simulation evidently are the need for storing the event traces which can become extremely large in size and the fact that trace-driven simulation does not allow for simulating all intricate system behavior, such as the effects of speculative instruction execution in microprocessors.

An example of a high-level, trace-driven MPSoC simulation environment is the Sesame system-level modeling and simulation framework (Pimentel et al. 2006; Erbas et al. 2007). Sesame is based on the aforementioned Y-chart methodology (Kienhuis et al. 2002), and accordingly it recognizes separate application and architecture models. The application models are explicitly mapped onto the architecture models by means of trace-driven simulation. The workload of an application is captured by instrumenting the application model – which is a parallel specification of the application – with annotations that describe the application's computational and communication actions at a coarse-grained level (typically at the level of the execution of entire functions). By executing this instrumented application model, these annotations cause the generation of traces of application events that subsequently drive the underlying architecture model. This architecture model – capturing the system resources and their constraints – then simulates the consequences of the consumed computation and communication events in terms of extra-functional system behavior (performance, power consumption, etc.). Figure 6 depicts Sesame's layered organization, illustrating the mapping of two multimedia applications (an MP3 encoder and video decoder) onto a bus-based MPSoC platform. A special mapping layer in Sesame, which can be seen as an abstract (real-time) operating system (RTOS) model, provides the scheduling of application events in the case multiple application processes are mapped onto a single processing resource.

Orthogonal to most of the (processor) simulation methods described above, there are additional techniques to further improve the simulation speed (Eeckhout 2010). In *sampled simulation*, for example, the simulation does not cover the execution of an entire program but only simulates relatively small samples of the program's execution. Here, the challenge is to select the samples in such a manner that they sufficiently represent the behavior as if the entire program was simulated. Another technique for speeding up simulation is *statistical simulation*. Rather than using real (benchmark) programs for simulation, it uses a statistical program profile. This profile captures the distributions of important program characteristics and is used for generating a synthetic instruction trace that drives a simple trace-driven simulator. As the synthetic trace is randomly generated from a statistical profile, this type of simulation can converge to a set of performance predictions fairly quickly.

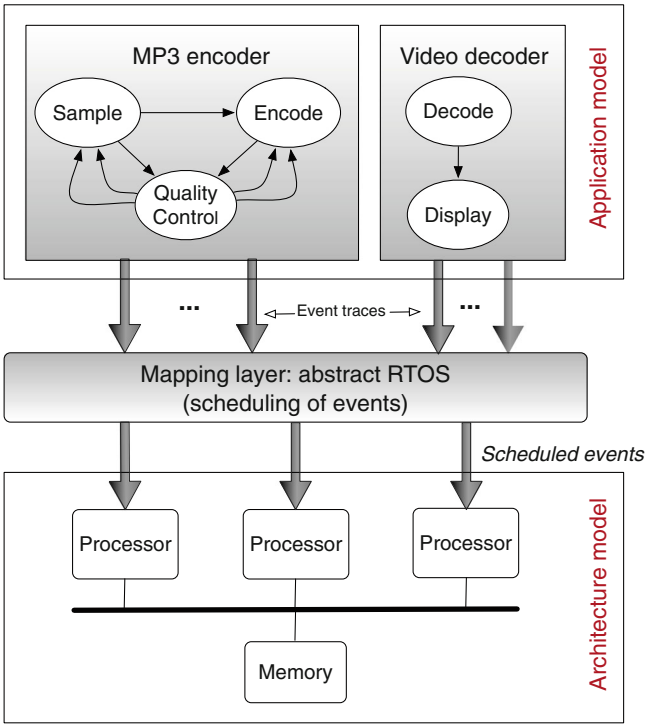


Fig. 6 The Sesame system-level MPSoC simulation infrastructure

Analytical Fitness Evaluation

In comparison to simulation, analytical models allow for much more efficient prediction of the extra-functional system behavior at the expense of a reduced accuracy. This makes analytical models very suitable for exploring large design spaces and to rapidly identify regions of interest that can be later explored in more detail using simulation. Another advantage of analytical models is that they can provide direct insight into the relationship between model parameters (representing design choices) and the predicted extra-functional behavior. For simulative methods, such understanding would require a large number of simulation runs.

Analytical models can roughly be divided into three classes (Eeckhout 2010): (1) *mechanistic* (or whitebox) models, (2) *empirical* (or blackbox) models, and (3) a hybrid combination of mechanistic and empirical modeling. Mechanistic models are based on first principles, which implies that they are built in a bottom-up fashion starting from a basic understanding of the mechanics of the modeled system. For example, in a mechanistic microprocessor performance model, penalties due to cache misses, branch mispredictions, the execution of instructions with different latencies, etc., are explicitly captured in the model.

In empirical models, statistical inference and machine learning techniques, like regression models or neural networks, are used to automatically synthesize a model through the process of learning from training data. For example, using a set of micro-architectural parameters such as pipeline depth, issue width, cache sizes, etc., one could train a model that predicts the instructions per cycle (IPC) or cycles per instruction (CPI) of a microprocessor. Inferring a model by means of automatic training typically is easier than developing a mechanistic model because it does not require intimate understanding of the mechanics of the modeled system. Evidently, the latter is also an immediate drawback as empirical models also tend to provide less insight than mechanistic models.

In hybrid mechanistic-empirical modeling, which is sometimes referred to as greybox modeling, extra-functional system aspects are captured using a formula that has been derived from insights in the underlying system. However, this formula includes a number of unknown parameters, which are then inferred through fitting (e.g., using regression), similarly to empirical modeling. Such hybrid mechanistic-empirical modeling is motivated by the fact that it provides insight (like mechanistic modeling) while easing the construction of the model (like empirical modeling).

Searching the Design Space

As explained before, searching a design space is a multi-objective optimization process. This process will evidently benefit from a good trade-off between speed, accuracy, and effort of the method for evaluating the fitness of a single design point, as discussed in the previous section. But, even if this trade-off is ideal, it should still be ensured that each evaluation of a design point contributes as much as possible to an effective and efficient search of the design space. A crucial component toward this goal is the search algorithm that navigates through the design space toward areas of interest by proposing which design points to evaluate next. Regardless of the specific type of search method that is used for such a design space traversal, its success depends on three major concerns, as was shown in Fig. 3: confidence, convergence, and effort. As was already mentioned earlier, these concerns typically cannot be considered in isolation, as they are highly interdependent, contradictory, and sometimes overlapping. The state of the art in DSE can be summarized as finding a good trade-off between these concerns.

DSE search algorithms can be divided into *exact* and *non-exact* methods. In exact DSE methods, like those implemented using integer linear programming (ILP) solutions (e.g., Niemann and Marwedel 1997; Lukasiewycz et al. 2008) or branch and bound algorithms (e.g., Padmanabhan et al. 2011), the optimum is guaranteed to be found. As such methods generally are compute intensive, they typically use design space pruning (i.e., discarding unsuitable design points) to optimize the efficiency of the search, thereby allowing them to handle larger design spaces. However, for realistic design problems with design spaces that are vast, these methods may still not scale and thus be less suited. Alternatively, in non-exact methods, *metaheuristics* are typically used to find a design point in the

known design space that meets the design requirements as best as possible. To this end, these methods search the design space for optimal solutions using only a finite number of design point evaluations and can thus handle larger design spaces. However, there is no guarantee that the global optimum will be found using metaheuristics, and therefore the result can be a local optimum within the design space. Examples of metaheuristics are hill climbing, tabu search, simulated annealing, ant colony optimization, particle swarm optimization, and genetic algorithms (GA) (Panerati et al. 2017). In this chapter, the focus will be on methods to navigate the design space that are based on GA. GA-based DSE has been widely studied in the domain of system-level embedded design (e.g., Palesi and Givargis 2002; Madsen et al. 2006; Erbas et al. 2006; Quan and Pimentel 2014; Goens et al. 2016) and has demonstrated to yield good results. Moreover, GAs can be used in their basic (domain-independent) form or, as will also be explained later on, with custom extensions that incorporate domain-dependent knowledge in order to improve search performance even further.

GA-Based DSE

GAs operate by searching through the solution space (spanned by the design variables/decisions being explored) where each possible solution is encoded as a string-like representation, often referred to as the *chromosome* (Beasley et al. 1993). A (randomly initialized) *population* of these chromosomes is then iteratively modified by performing a fixed sequence of actions that are inspired by their counterparts from biology: fitness evaluation and selection, crossover, and mutation. A fundamental design choice of a GA is the genetic representation of the solution space, because each of the crossover and mutation steps depends on it. To illustrate how such a genetic representation could look like, let us use a widely studied DSE problem in the domain of system-level embedded systems design as an example: optimizing the mapping of a (set of) concurrent application(s) onto an underlying (heterogeneous) MPSoC platform architecture (Singh et al. 2013). As a convenient mapping description for an application with n tasks, a vector of size n is used with processor identifiers p_i, where p_i indicates the mapping target of task i:

$$[p_0, \ldots, p_i, \ldots, p_{n-1}]$$

This commonly used description is very suitable to serve as the chromosome representation for a GA. A valid mapping specification is a *feasible* partitioning of all n tasks. In this context, "feasible" means that tasks are mapped onto processing elements that *can* execute those tasks (i.e., there are no functional restrictions of the processing element in question, like an ASIC component which only allows the execution of one particular piece of functionality) and that communicating tasks are mapped onto processing elements that can actually communicate with each other (i.e., there are no topological communication restrictions). In case an infeasible mapping is created by the genetic operators of a GA (crossover and mutation), a

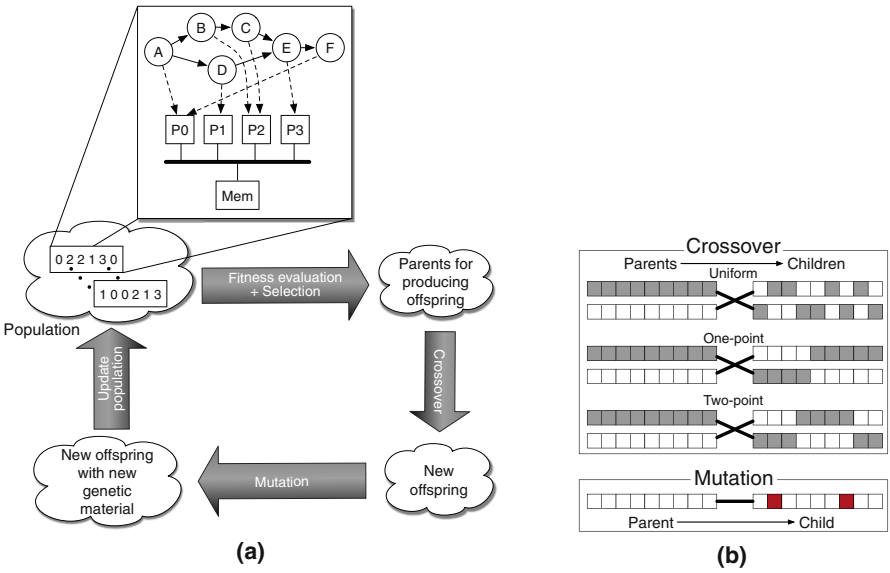


Fig. 7 GA-based mapping DSE: (**a**) general overview of the GA steps and (**b**) crossover and mutation operators

mechanism is required that either discards or repairs such a chromosome. Repairing a chromosome implies that it is transformed into a valid chromosome (mapping) that is "as close as possible" to the original, invalid one. Moreover, note that task partitions specifying a mapping may also be empty (i.e., particular processor(s) not in use) or contain all n tasks (i.e., a single processor system). A processor that is not assigned any tasks (having an empty task partition) can be considered idle or nonexistent.

In Fig. 7a, the different steps of a GA are shown. This figure also illustrates the mapping representation of a chromosome for an application with six tasks and a four-processor bus-based MPSoC platform. Starting from a (randomly initialized) population of chromosomes, representing the different mapping design instances, the fitness of the mapping solutions in the population is first evaluated. To this end, any of the analytical or simulative techniques discussed in section "Evaluation of a Single Design Point" can be used. Subsequently, based on the fitness evaluation, a selection of chromosomes is made that will be used to create offspring. This offspring is created by combining genetic material from two parents using a crossover operation, as illustrated in the top part of Fig. 7b. There exist various forms of this crossover operator, of which the uniform, one-point, and two-point crossovers are the most popular. Next, new genetic material is introduced in the offspring by means of a mutation operator as illustrated at the bottom of Fig. 7b. Such a mutation randomly changes one or more genes within chromosomes. Finally, the newly created offspring is used to update the population by either replacing it

or by deploying so-called elitism. Such elitism involves the combination of the new offspring with a small number of the best solutions from the original population to avoid loosing strong solutions.

To provide a small example of the results a GA-based DSE could obtain, some results are presented of a small-scale case study where the design space consists of an application with 11 tasks that is to be mapped onto a four-core MPSoC architecture with a crossbar interconnect (Thompson 2012). The mapping design space contains more than 4 million design points. Of these design points, 175K are unique ones since the target platform is a homogeneous, symmetric MPSoC and, as a consequence, exhibits mapping symmetries. Because of the relatively small design space, in this particular case, it was also possible to perform an exhaustive search, allowing a quality evaluation of the GA-based search results. To account for the stochastic behavior of GAs, all results are averages over 300 GA runs. The fitness of mapping solutions has been evaluated using the Sesame MPSoC simulation framework (Pimentel et al. 2006; Erbas et al. 2007) (see also section "Simulative Fitness Evaluation"). Figure 8 shows the results of the GA-based DSE with different population sizes (10, 15, 40, or 80 chromosomes), a constant mutation rate (0.1) and crossover probability (0.9), and a uniform crossover in a so-called P-Q (probability-quality) plot. Regarding the top part of this plot, the horizontal axis (top x-axis) represents the quality of the result as a percentile toward the true optimum (a lower percentile indicates a result closer to the optimum), and the vertical axis represents the probability of achieving a result with that quality. The straight lines in the graph represent the theoretically derived probabilities of finding results using a simple, uniform random search. The 80–95% confidence intervals of the mean fitness value (execution time in cycles, in this case) of mapping solutions found by the GA were also computed, averaged over the 300 runs of each GA search. These

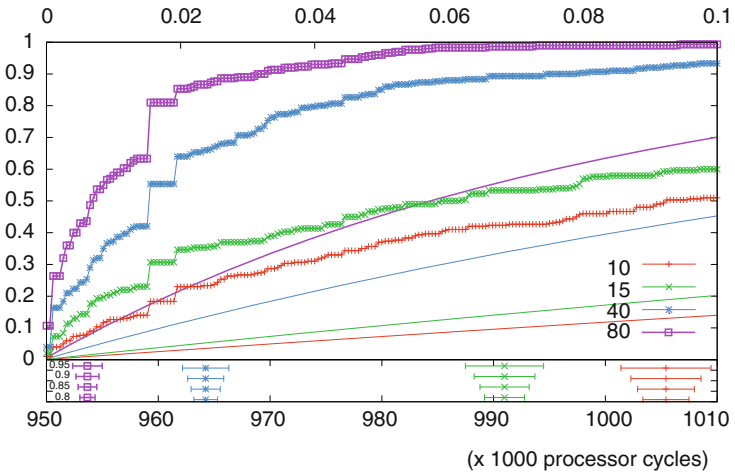


Fig. 8 P-Q plot for GA-based DSE with different population sizes

confidence intervals, shown at the bottom of the graph in Fig. 8, indicate the degree of certainty (as specified by the confidence level) that the real mean lies within the confidence interval. The more the confidence intervals for different experiments are nonoverlapping, the more significant the difference of the mean behavior (which is clearly the case in the example of Fig. 8). The results from this particular case study show that the GA-based DSE with the largest population size can find mapping solutions that are always very close to the real optimum: within the 0.1 percentile, implying that they belong to the best $\frac{175K}{1000} = 175$ solutions. A larger population size, however, comes with a higher number of fitness evaluations during the search and thus requires a longer search time (assuming the number of search iterations remains constant). According to Fig. 8, a population size of 40 may therefore provide a good compromise.

Optimizing GA-Based DSE

There are various methods for making the search process of a GA-based DSE more efficient. This allows the DSE process to either find the design candidates quicker (i.e., improve the convergence behavior of the DSE) or to spend the same amount of time to evaluate more design points. The latter can be used to enable the search of larger design spaces or to improve the chance of finding better design candidates (i.e., improve the confidence property of the DSE). One approach for optimizing the GA-based search is to enrich the genetic operators of the GA with domain knowledge such that they produce more diverse offspring or offspring with a higher probability of being closer to the optimum. For example, in Thompson and Pimentel (2013), new GA operators have been proposed that optimize the search performance by (1) reducing the redundancy present in chromosome representations (e.g., mapping symmetries (Goens et al. 2017) in the case of homogeneous, symmetrical MPSoC platforms) or (2) using a new crossover operator that is based on a mapping distance metric that provides a measure of similarity between mappings. Using this mapping distance information, the new crossover operator aims at retaining the strong chromosome parts of both of the parents. In Quan and Pimentel (2014), a new mutation operator has been proposed that considers the affinity of tasks with respect to processors, the communication cost between tasks, and the differences of processor workloads to steer the mutation in such a way that offspring is produced with a higher probability of being (near-)optimal.

Another approach for optimizing GA-based DSE concerns the reduction of the time taken to evaluate the fitness of solutions during the GA's execution. As mentioned before, DSE approaches typically use either simulation or an analytical model to evaluate the fitness of design points, where simulative approaches prohibit the evaluation of many design options due to the higher evaluation performance costs and analytical approaches suffer from accuracy issues. Therefore, in Piscitelli and Pimentel (2012a), a hybrid form of mapping DSE has been proposed that combines simulation with analytical estimations to prune the design space in terms of application mappings that need to be evaluated using simulation. To this end,

the DSE technique uses an analytical model that estimates the expected throughput of an application given a certain architectural configuration and application-to-architecture mapping. In the majority of the search iterations of the DSE process, this analytical throughput estimation avoids the use of simulation to evaluate the design points. However, since the analytical estimations may in some cases be less accurate, the analytical estimations still need to be interleaved with simulative evaluations in order to ensure that the DSE process is steered into the right direction (Piscitelli and Pimentel 2012b). A similar approach is taken in Mariani et al. (2010), where an iterative DSE methodology is proposed exploiting the statistical properties of the design space to infer, by means of an empirical analytic model, the design points to be analyzed with low-level simulation. The knowledge of a few design points is used to predict the expected improvement of unknown configurations.

Alternatively, in hierarchical DSE (e.g., Mohanty et al. 2002; Jia et al. 2013, 2014), DSE is first performed using analytical or symbolic models to quickly find the interesting parts in the design space. Hereafter, simulation-based DSE is performed on the selected sweet spots in the design space to more accurately search for the optimal design points.

Multi-application Workload Models

The DSE techniques discussed so far focus on the evaluation and exploration of MPSoC architectures under static, single-application workloads. Today's MPSoC systems, however, often require supporting an increasing number of applications and standards, where multiple applications can run simultaneously and concurrently contend for system resources (Thompson and Pimentel 2007; Castrillon et al. 2013). For each single application, there may also be different execution modes (or program phases) with different computational and communication requirements. For example, in software-defined radio appliances, a radio may change its behavior according to resource availability, such as the long-term evolution (LTE) standard which uses adaptive modulation and coding to dynamically adjust modulation schemes and transport block sizes based on channel conditions. Or a video application could dynamically lower its resolution to decrease its computational demands in order to save battery life. As a consequence, the behavior of application workloads executing on the embedded system can change dramatically over time.

As illustrated in Fig. 9, there are several approaches for dealing with multi-application workloads in the context of DSE. A commonly used approach is to consider the applications in isolation, as illustrated in Fig. 9a. This implies that each of the applications in the multi-application workload will be mapped to a different, isolated part of the system. As a consequence, the DSE for each of these applications can also be performed in isolation. However, this approach typically leads to overdesigned systems since there is no or limited resource sharing between applications. Another approach, illustrated in Fig. 9b, makes the pessimistic assumption that all applications that can be executed on the system will always be active (and will thus be contending for system resources). Again,

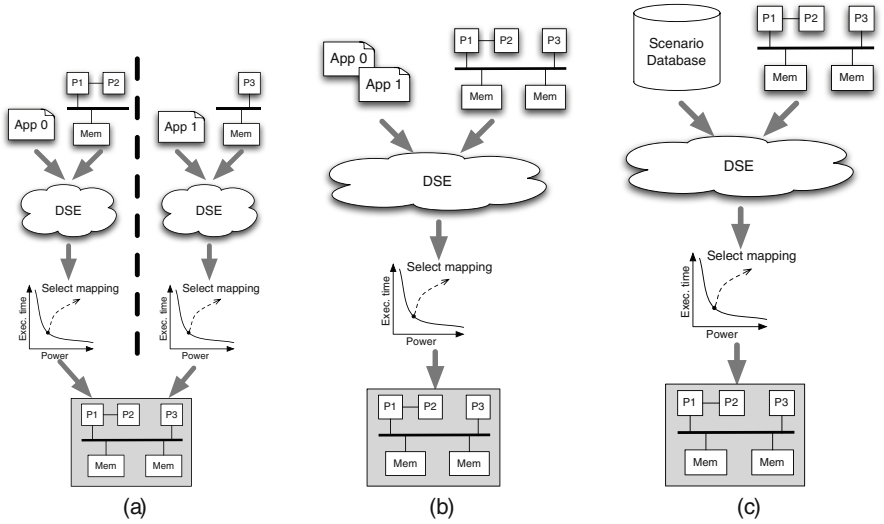


Fig. 9 DSE for multi-application workloads on a three-core, bus-based MPSoC: (**a**) DSE with application isolation, (**b**) pessimistic DSE, and (**c**) scenario-based DSE

performing DSE with such an assumption may lead to highly overdesigned systems, as in reality the concurrent activation of all possible applications may be unlikely. To address the problem of overdesigning systems and to capture the dynamism in application workload behavior during the design process, the DSE could employ the concept of application scenarios (Gheorghita et al. 2009), leading to scenario-based DSE (van Stralen and Pimentel 2010a, 2013; Pimentel and van Stralen 2017; Castrillon et al. 2013). This is illustrated in Fig. 9c. The remainder of this section will discuss the concepts of application scenarios and scenario-based DSE, again using the example of application mapping exploration for illustration purposes.

Scenario-Based DSE

Application scenarios are able to describe the dynamism of embedded applications and the interaction between the different applications on the embedded system. An application scenario consists of two parts: an inter- and an intra-application scenario. An inter-application scenario describes the interaction between multiple applications, i.e., which applications are concurrently executing at a certain moment in time. Inter-application scenarios can be used to prevent the overdesign of a system. If some of the applications cannot run concurrently, then there is no need of reserving resources for the situation where these applications are running together. Intra-application scenarios, on the other hand, describe the different execution modes for each individual application. The concept of application scenarios, inter-application scenarios, and intra-application scenarios is illustrated in Fig. 10 for

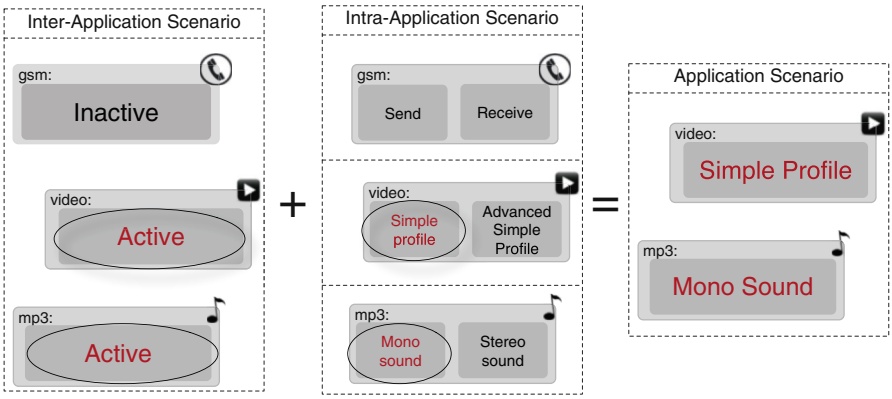


Fig. 10 Application scenarios, inter-application scenarios, and intra-application scenarios

three multimedia applications (mp3 player, video decoder, and gsm application) with each two application modes.

The number of different application scenarios grows exponentially with the number of applications involved. So, to perform DSE with these application scenarios, scenario-based DSE needs to solve the problem that the total number of possible application scenarios is too large to exhaustively evaluate the fitness of design points with *all* of these scenarios. Therefore, a small but *representative subset of application scenarios* must be selected for the evaluation of MPSoC design points. This representative subset must be used for comparing mappings and should lead to the same performance ordering as would have been produced when the complete set of the application scenarios would have been used. That is, if mapping *m1* is better than mapping *m2*, the representative subset should be able to give a better predicted fitness to mapping *m1* than it assigns to mapping *m2*. However, the selection of such a representative subset is not trivial (Pimentel and van Stralen 2017). This is because the representative subset is dependent on the current set of mappings that are being explored. Depending on the set of mappings, a different subset of application scenarios may reflect the relative mapping qualities of the majority of the application scenarios.

As a result, the representative subset cannot be statically selected. For a static selection, one would need to have a large fraction of the mappings that are going to be explored during the MPSoC DSE. However, since these mappings are only available during DSE, a dynamic selection method must be used. Thus, both the set of optimal mappings and the representative subset of scenarios need to be *co-explored simultaneously* such that the representative subset is able to adapt to the set of mappings that are currently being explored (van Stralen and Pimentel 2010a, 2013; Pimentel and van Stralen 2017). Figure 11 shows the scenario-based DSE framework. The left part of the picture provides a general overview of the exploration flow, whereas the right part illustrates the scenario-based DSE in more detail. As input, the scenario-based DSE requires a database of application

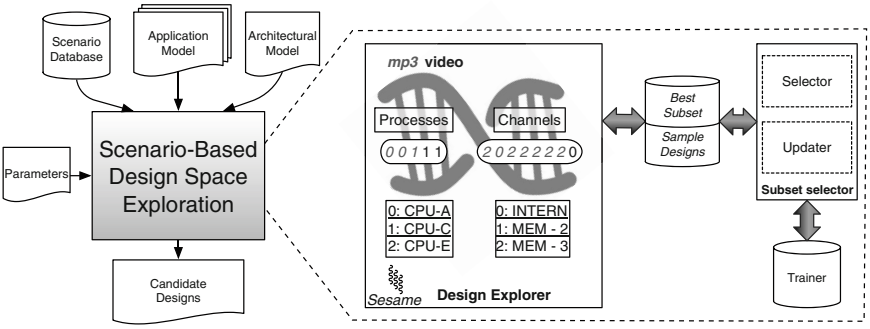


Fig. 11 The exploration framework for scenario-based DSE

scenarios, application models, and an MPSoC platform architecture model. The description of the application workload is split into two parts: (1) the structure and (2) the behavior. The structure of applications is described using application models (as described before), whereas a scenario database (van Stralen and Pimentel 2010b) explicitly stores all the possible multi-application workload behaviors in terms of application scenarios (i.e., intra- and inter-application scenarios). In the scenario-based DSE framework, two separate components are recognized that simultaneously perform the co-exploration tasks: the design explorer searches for the set of optimal mappings, while the subset selector tries to select a representative subset of scenarios. To this end, they exchange data in an asynchronous fashion after every search iteration. Here, the design explorer sends a sample of the current mapping population to the subset selector, whereas the subset selector makes the most representative subset available for the fitness prediction in the design explorer.

The design explorer performs a traditional mapping DSE using a GA, as discussed in section "Searching the Design Space". As explained above, it uses a representative subset of scenarios to evaluate the fitness of mapping solutions. At every iteration of the GA, the design explorer reads in the most recent representative scenario subset from the subset selector and submits the current population of mapping solutions to the subset selector in order to allow the latter to select the appropriate representative subset. This subset selection is not trivial as there are many scenarios to pick from, leading to a huge number of possible scenario subsets. Therefore, the subset selector uses the set of mappings it regularly receives from the design explorer to train the scenario subset such that it is representative for the current population in the design explorer. As the population of the design explorer slowly changes over time, the representative subset will change accordingly. In van Stralen and Pimentel (2013), three different techniques for selecting a representative scenario subset are presented and evaluated: a GA-based scenario space search (which means that two GAs are running concurrently, one for the design explorer and one for the subset selector), a feature selection (FS)-based search algorithm, and a hybrid combination (HYB) of these two. The latter aims at combining the strengths of both the GA-based and FS-based searches. That is, a GA is capable of

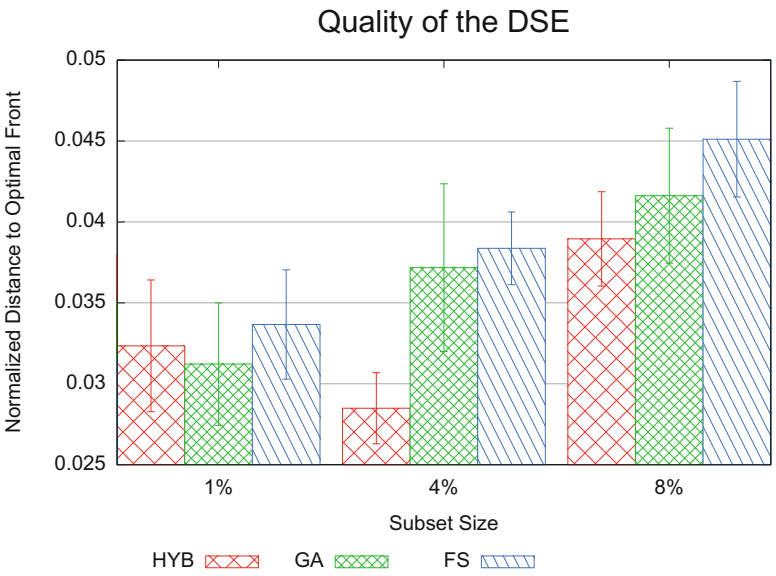


Fig. 12 Quality of the scenario-based DSE for the different subset selection approaches. The quality is determined based on the distance between the estimated Pareto front and the optimal front

quickly exploring the space of potential scenario subsets, but due to its stochastic nature, it is susceptible to missing the optimal scenario subsets. This is not the case with the FS algorithm as it more systematically explores the local neighborhood of a scenario subset.

To give a feeling of the performance of the three different fitness prediction techniques, Fig. 12 shows the results of a scenario-based DSE experiment in which the three techniques are compared for three different scenario subset sizes: 1%, 4%, and 8% of the total number of application scenarios. In this experiment, the mapping of ten applications with a total of 58 tasks and 75 communication channels is explored. The multi-application workload consists of 4607 different application scenarios in total. The target platform is a heterogeneous MPSoC with four general-purpose processors, two ASIPs and two ASICs, all connected using a crossbar network. In this experiment, a DSE with a fixed duration of 100 min is performed for all three subset selector approaches. The results have been averaged over nine runs. To evaluate the fitness of mapping solutions, the Sesame MPSoC simulation framework (see section "Simulative Fitness Evaluation") is again deployed. To determine the efficiency of the multi-objective DSE, the distance of the estimated Pareto front (execution time versus energy consumption of mapping solutions) to the optimal Pareto front is obtained. For this purpose, the execution time and energy consumption are normalized to a range from 0 to 1. As the optimal Pareto front is not exactly known since the design space is too large to be exhaustively searched, the combined Pareto front of all performed experiments is used for this.

The size of the scenario subset provides a trade-off between accuracy and convergence of the search. That is, a larger scenario subset will lead to a more accurate fitness prediction of mappings in the design explorer at the cost of a larger computational overhead to obtain the fitness of a single mapping causing a slower convergence of the search. This can be seen in Fig. 12. The GA and the FS subset selection methods have worse results when the subset becomes larger (remember that a fixed DSE duration of 100 min is used). For a subset size of 4%, the hybrid selector is, however, still able to benefit from a subset with a higher accuracy. The slower convergence only starts to effect the efficiency for the 8% subset. Comparing the different methods, the hybrid method shows the best results. The only exception is for the 1% subset. In this case, the GA is still able to search the smaller design space of possible subsets. Still, the result of the hybrid method at 4% is better than the result of the GA at 1%. With the larger subset sizes, the hybrid method can exploit both the benefits of the FS and the GA.

Application Exploration

As described in section "Y-Chart-Based DSE", the Y-chart methodology is a popular approach for system-level DSE in the domain of embedded systems. This means that exploration can take place to investigate (the fitness of) different MPSoC architectures and different mappings of application tasks to the underlying architecture but also different application implementations. With respect to the latter, one could, for example, explore the use of different algorithms for implementing a certain piece of application functionality or vary the degree of concurrency in an application (e.g., fine-grained versus more coarse-grained concurrency) that can be exploited by an underlying MPSoC platform. So far, the discussion was limited to the first two types of exploration, i.e., exploring different MPSoC architectures and task-to-architecture mappings. In this section, an example of application exploration will be described, and this will be done for the popular application domain of deep learning. More specifically, an approach will be outlined for so-called neural architecture search (NAS) (Sapra and Pimentel 2020a,b), which automates the discovery of an efficient neural network for a given task, such as image/video recognition, classification, natural language processing, etc.

NAS by Means of Evolutionary Piecemeal Training (EPT)

The NAS approach that will be described searches for an efficient convolutional neural network (CNN) architecture. To this end, it leverages a GA, which allows a group of candidate CNNs in the GA's population to train in parallel. In most NAS techniques, training of a neural network is considered a separate task or a performance estimation strategy to perform the neural network architecture search. However, the approach described here considers NAS from a different perspective as it aims at finding optimal CNN architectures *during the training process itself*

as opposed to accuracy prediction or training as a separate performance estimation strategy. The NAS approach is called EPT (Sapra and Pimentel 2020a,b), where piecemeal training refers to training a neural network with a small "data piece" of size δ_k. In this technique, a traditional continuous training is interceded by an evolutionary operator at regular intervals, and the interval of intervention is dictated by the value of δ_k. The evolutionary operators applied to CNNs in the GA population lead to CNN architecture modifications and hence exploration of the search space. A new CNN architecture derived like this is always partially trained already as it was modified from another CNN undergoing training. In subsequent iterations, derived CNN architectures continue to train. Those CNN candidates that are not able to achieve high accuracy during training will be dropped from the population. This can also be seen as early training termination of candidates that are performing poorly. Toward the end of this algorithm, the best candidates are selected from the population, which can then be post processed or trained further, if needed.

The search space for the algorithm is focused on plain CNNs, which consist of convolutional, fully connected (FC) and pooling layers without residual connections, branching, etc. Batch normalization and nonlinear activations are configurable and can be added to the network. CNN architectures are defined by a group of blocks, where each block is of a specific layer type and is bounded by minimum and maximum number of layers it can have. Additionally, each layer has upper and lower bounds on each of its parameters. For example, a convolutional layer will have bounds on the number of units, kernel sizes, and stride sizes possible. These constraints are in place to make sure that CNN architectures do not become too big and limit the resource consumption of the final neural network. This is an important factor to consider when mapping the CNNs to resource-constrained embedded systems. The search space specifications along with its bounds are encoded as a collection of genes, also called a genotype. All possible combinations of parameters together form the gene pool from which individual neural networks are created and trained.

A population-based training process is used where an initial population of neural networks is randomly created from the defined gene pool. In each iteration, all candidates of the population are piecemeal-trained and then evaluated using the validation set. Depending on the available resources, all candidates can be trained in any combination of parallel and sequential manner. The size of the population is kept constant throughout the algorithm, though the candidates of the population keep changing as they are altered through the evolutionary operators applied in each iteration. The number of candidates in the population needs to be large enough to maintain enough diversity of CNN architectures in the population while still satisfying the constraints applied to it.

Evolutionary Operators

The crossover operator in the EPT-based NAS works with two neural networks and swaps all layers in a gene block of the same type. In this replacement, the layers being swapped are roughly in the same phase of feature extraction. The input and output feature map sizes of the layer block being swapped are also identical

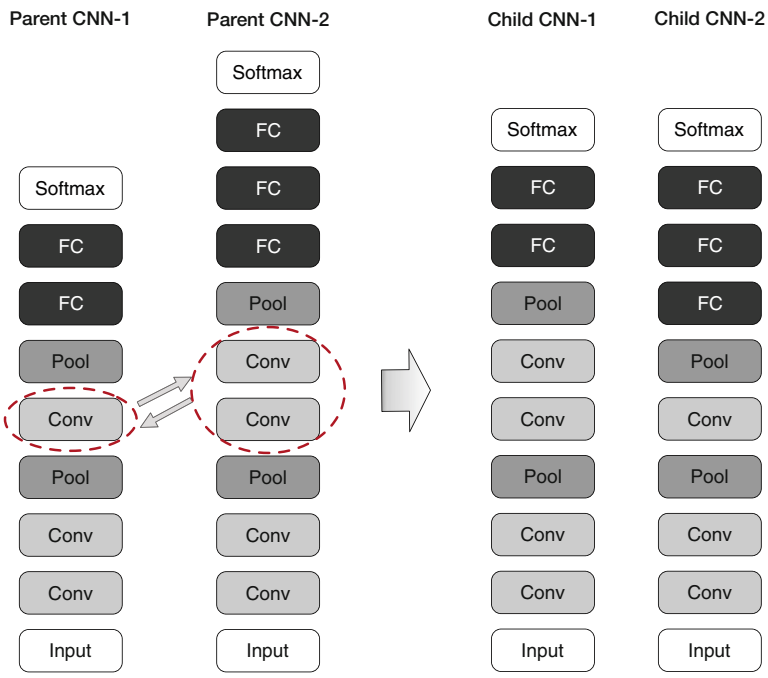


Fig. 13 Crossover operator on two CNNs swapping convolution layers

in both of the selected networks. Figure 13 illustrates the crossover operator for swapping convolutional layers from two networks. Crossover is not a function preserving operator, but in experiments they were found to be important to introduce diversity in the population by changing the total number of layers in a candidate through swapping. To reduce the negative effect of training loss incurred due to the crossover, a cooling-down approach is used to the crossover rate. In earlier GA iterations, where the training loss is already high, there are more swaps happening than in the later ones, where training loss is very low.

The mutation operator changes a layer's parameters such as the number of kernels or kernel size and is designed to be function preserving. Every mutation disrupts the ongoing training of the mutated candidates, and some additional loss is incurred in the training in process. However, due to the function preserving nature of the mutation operator, the loss incurred from this operator is as small as possible and recoverable in later piecemeal training.

NAS Results

To illustrate the competence and versatility of the EPT-based NAS concept, a range of experiments were performed with datasets from two different domains: CIFAR-10 for image classification (Deng et al. 2009) and PAMAP2 for human activity recognition (Reiss and Stricker 2012). For CIFAR-10, the search took

Table 1 CIFAR-10 Accuracy comparisons with evolutionary approaches

Model	Search space	GPU-days	Accuracy(%)
CoDeepNeat (Miikkulainen et al. 2019)	Hybrid	–	92.7
GeneticCNN (Xie and Yuille 2017)	Hybrid	17	92.9
EANN-Net (Chen et al. 2019)	Hybrid	–	92.95
AmoebaNet (Real et al. 2019)	Cell	3150	**96.6**
NSGANet (Lu et al. 2018)	Hybrid	8	96.15
Evolution (Real et al. 2017)	Hybrid	1000+	94.6
EPT	Plain CNN	**2**	92.5

2-GPU days, and the best prediction accuracy was found to be 92.5% on the test set. Table 1 shows comparisons with other evolutionary NAS approaches, where EPT refers to evolutionary piecemeal training. It may seem that 92.5% is relatively low compared to other published works, but this is on a very simple and plain CNN without any architectural enhancements or advance data augmentation. Other approaches use a hybrid search space where different architecture blocks or cell modules as well as arbitrary skip connections are used. Instead of stacking conventional layers, these stack different blocks. The best model found in the EPT experiments has 13 convolutional layers followed by two fully connected layers. For the PAMAP2 dataset, the EPT search took only 10 GPU-hours, and the best prediction accuracy was 94.36%. Compared to state-of-the-art neural network solutions for this particular dataset, EPT outperforms all other known efforts. The best performance was found on a neural network that has seven convolutional layers followed by three fully connected layers. The interested reader is referred to Sapra and Pimentel (2020a) for a more detailed analysis of EPT's experimental results.

Conclusion and Outlook

In this chapter, an overview was presented of techniques and methods for DSE of embedded systems. The discussion was organized along the lines of the two primary elements of DSE: the evaluation of single design points and the search strategy for covering the design space. The overview is certainly not meant to be exhaustive. For example, the discussion mainly focused on popular GA-based DSE, optimizing system performance and, to some extent, power/energy consumption. The optimization of other important design objectives, such as system reliability (e.g., addressed in Jhumka et al. 2005; Glaß et al. 2007, 2008; van Stralen and Pimentel 2012), has not been covered.

There are still many open research challenges for this domain. For example, embedded systems more and more need to become adaptive systems due to increasingly dynamic application workload behavior (as was previously discussed); the need for quality-of-service management to *dynamically* trade off different system qualities such as performance, precision, and power consumption; and the

fact that a technology level is reached where digital circuits are no longer fully reliable, increasing the chances of transient and permanent faults. This calls for research to take system adaptivity, in which a system can continuously reconfigure and customize itself at run time according to the application workload at hand and the state of the system (e.g., Singh et al. 2013; Quan and Pimentel 2015, 2016a,b; Goens et al. 2017; Khasanov and Castrillon 2020), into account in the process of DSE. In the case of adaptive systems, a DSE process cannot easily compare different design choices by, e.g., simply evaluating the performance or power/energy consumption of an application workload executing on a specific platform architecture. That is, the reconfiguration behavior (i.e., when and how the system reacts to "disruptive events" that trigger system reconfigurations) of the system and the performance/power consumption consequences of such system adaptivity actions must be taken into account when comparing different design instances. This calls for efficient and effective methods that allow for evaluating and optimizing adaptive embedded systems designs such that the way the system instances and their extra-functional behavior evolve over time is also captured.

Another research direction that is worth mentioning involves the introduction of new design objectives in the process of (early) DSE, in addition to the traditional objectives such as system performance, power/energy consumption, system cost, and reliability. Arguably, a good example is the need for taking system security into account as an optimization objective (Pimentel 2020). As embedded systems are becoming increasingly ubiquitous and interconnected, they attract a worldwide attention of attackers, which makes the security aspect more important than ever during the design of those systems. Currently, system security is still mostly considered as an afterthought and is typically not taken into account during the very early design stages. However, any security measure that may eventually be taken later in the design process does affect the already established trade-offs with respect to the other system objectives such as performance, power/energy consumption, cost, etc. Thus, covering the security aspect in the earliest phases of design is necessary to design systems that are, in the end, optimal with regard to all system objectives. However, this poses great difficulties because unlike the earlier mentioned conventional system objectives like performance and power consumption, security is hard to quantify. This necessitates research on techniques that make it possible to incorporate security as an objective in early DSE.

At this moment, the integration of security aspects in the process of system-level DSE of embedded systems is still a largely uncharted research ground. Only a few efforts exist that address this problem, but they typically provide only partial solutions or solutions to very specific security problems (e.g., Lin et al. 2015; Weichslgartner et al. 2016; Stierand et al. 2014; Tan et al. 2017). Moreover, in most of these works, security is modeled as a requirement in the DSE process, which does not allow for studying actual trade-offs between performance, power consumption, and cost in relationship to secureness of a design. Only a handful of research efforts, such as Ferrante et al. (2013) and Gressl et al. (2019), seem to have been aiming at incorporating security as an objective that can be traded off with other objectives during early DSE.

References

Balarin F, Sentovich E, Chiodo M, Giusto P, Hsieh H, Tabbara B, Jurecska A, Lavagno L, Passerone C, Suzuki K, Sangiovanni-Vincentelli A (1997) Hardware-software co-design of embedded systems – the POLIS approach. Kluwer Academic Publishers, Norwell

Beasley D, Bull DR, Martin RR (1993) An overview of genetic algorithms: part I-fundamentals. Univ Comput 15(2):58–69

Bellard F (2005) Qemu, a fast and portable dynamic translator. In: Proceedings of the USENIX annual technical conference, pp 41–46

Binkert N et al (2011) The gem5 simulator. SIGARCH Comput Archit News 39(2):1–7

Bringmann O, Ecker W, Gerstlauer A, Goyal A, Mueller-Gritschneder D, Sasidharan P, Singh S (2015) The next generation of virtual prototyping: ultra-fast yet accurate simulation of hw/sw systems. In: Proceedings of the international conference on design, automation & test in Europe (DATE), pp 1698–1707

Butko A, Garibotti R, Ost L, Lapotre V, Gamatie A, Sassatelli G, Adeniyi-Jones C (2015) A trace-driven approach for fast and accurate simulation of manycore architectures. In: Proceedings of the Asia and South Pacific design automation conference (ASP-DAC), pp 707–712

Cai L, Gajski D (2003) Transaction level modeling: an overview. In: Proceedings of the international conference on hardware/software codesign and system synthesis (CODES+ISSS), pp 19–24

Castrillon J, Velasquez R, Stulova A, Sheng W, Ceng J, Leupers R, Ascheid G, Meyr H (2010) Trace-based KPN composability analysis for mapping simultaneous applications to MPSoC platforms. In: Proceedings of the conference on design, automation test in Europe (DATE), pp 753–758

Castrillon J, Leupers R, Ascheid G (2013) MAPS: mapping concurrent dataflow applications to heterogeneous MPSoCs. IEEE Trans Ind Inf 9(1):527–545

Ceng J, Sheng W, Castrillon J, Stulova A, Leupers R, Ascheid G, Meyr H (2009) A high-level virtual platform for early MPSoC software development. In: Proceedings of the 7th IEEE/ACM international conference on hardware/software codesign and system synthesis (CODES+ISSS)

Chen Z, Zhou Y, Huang Z (2019) Auto-creation of effective neural network architecture by evolutionary algorithm and resnet for image classification. In: 2019 IEEE international conference on systems, man and cybernetics (SMC). IEEE, pp 3895–3900

Deng J, Dong W, Socher R, Li LJ, Li K, Fei-Fei L (2009) Imagenet: a large-scale hierarchical image database. In: 2009 IEEE conference on computer vision and pattern recognition. IEEE, pp 248–255

Eeckhout L (2010) Computer architecture performance evaluation methods. Synthesis lectures on computer architecture. Morgan & Claypool Publishers, San Rafael

Erbas C, Cerav-Erbas S, Pimentel AD (2006) Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design. IEEE Trans Evolut Comput 10(3):358–374

Erbas C, Pimentel AD, Thompson M, Polstra S (2007) A framework for system-level modeling and simulation of embedded systems architectures. EURASIP J Embed Syst 207:1–11

Ferrante A, Milosevic J, Janjšević M (2013) A security-enhanced design methodology for embedded systems. In: Proceedings of the international conference on security and cryptography (SECRYPT), pp 1–12

Gheorghita SV et al (2009) System-scenario-based design of dynamic embedded systems. ACM Trans Des Autom Electronic Syst 14(1):1–45

Glaß M, Lukasiewycz M, Streichert T, Haubelt C, Teich J (2007) Reliability-aware system synthesis. In: Proceedings of the conference on design, automation test in Europe, pp 1–6

Glaß M, Lukasiewycz M, Reimann F, Haubelt C, Teich J (2008) Symbolic reliability analysis and optimization of ecu networks. In: Proceedings of the conference on design, automation and test in Europe, pp 158–163

Goens A, Khasanov R, Castrillon J, Polstra S, Pimentel AD (2016) Why comparing system-level MPSoC mapping approaches is difficult: a case study. In: Proceedings of the IEEE 10th international symposium on embedded multicore/many-core systems-on-chip (MCSoC)

Goens A, Khasanov R, Castrillon J, Hähnel M, Smejkal T, Härtig H (2017) Tetris: a multi-application run-time system for predictable execution of static mappings. In: Proceedings of the 20th international workshop on software and compilers for embedded systems (SCOPES), pp 11–20

Goens A, Siccha S, Castrillon J (2017) Symmetry in software synthesis. ACM Trans Archit Code Optim 14(2):1–26

Gressl L, Steger C, Neffe U (2019) Security driven design space exploration for embedded systems. In: Forum for specification and design languages (FDL), pp 1–8

Gries M (2004) Methods for evaluating and covering the design space during early design development. Integr VLSI J 38(2):131–183

Jhumka A, Klaus S, Huss SA (2005) A dependability-driven system-level design approach for embedded systems. In: Proceedings of the conference on design, automation and test in Europe, pp 372–377

Jia ZJ, Bautista T, Núñez A, Thompson M, Pimentel AD (2013) A system-level infrastructure for multi-dimensional MP-SoC design space co-exploration. ACM Trans Embed Comput Syst 13:1–26(1s)

Jia ZJ, Núñez A, Bautista T, Pimentel AD (2014) A two-phase design space exploration strategy for system-level real-time application mapping onto MPSoC. Microprocessors Microsyst 38(1):9–21

Keutzer K, Newton A, Rabaey J, Sangiovanni-Vincentelli A (2000) System-level design: orthogonalization of concerns and platform-based design. IEEE Trans Comput-Aided Des Integr Circuits Syst 19(12):1523–1543

Khasanov R, Castrillon J (2020) Energy-efficient runtime resource management for adaptable multi-application mapping. In: Proceedings of the design, automation and test in Europe conference (DATE)

Kienhuis B, Deprettere FE, van der Wolf P, Vissers K (2002) A methodology to design programmable embedded systems: the Y-chart approach. Lect Notes Comput Sci Embed Process Des Chall 2268:18–37

Li S et al (2013) The McPAT framework for multicore and manycore architectures: simultaneously modeling power, area, and timing. ACM Trans Archit Code Optim 10(1):5

Lin CW, Zheng B, Zhu Q, Sangiovanni-Vincentelli A (2015) Security-aware design methodology and optimization for automotive systems. ACM Trans Des Autom Electron Syst 21(1):1–26

Lu Z, Whalen I, Boddeti V, Dhebar Y, Deb K, Goodman E, Banzhaf W (2018) NSGA-Net: a multi-objective genetic algorithm for neural architecture search. arXiv preprint arXiv:1810.03522

Lukasiewycz M, Glass M, Haubelt C, Teich J (2008) Efficient symbolic multi-objective design space exploration. In: Proceedings of the Asia and South Pacific design automation conference (ASP-DAC), pp 691–696

Madsen J, Stidsen TK, Kjaerulf P, Mahadevan S (2006) Multi-objective design space exploration of embedded system platforms. In: Kleinjohann B, Kleinjohann L, Machado RJ, Pereira CE, Thiagarajan PS (eds) From model-driven design to resource management for distributed embedded systems. Springer, Boston, pp 185–194

Mariani G, Brankovic A, Palermo G, Jovic J, Zaccaria V, Silvano C (2010) A correlation-based design space exploration methodology for multi-processor systems-on-chip. In: Proceedings of the design automation conference (DAC), pp 120–125

Miikkulainen R, Liang J, Meyerson E, Rawal A, Fink D, Francon O, Raju B, Shahrzad H, Navruzyan A, Duffy N et al (2019) Evolving deep neural networks. In: Artificial intelligence in the age of neural networks and brain computing. Elsevier, Amsterdam, pp 293–312

Mohanty S, Prasanna VK, Neema S, Davis J (2002) Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation. In: Proceedings of LCTES+SCOPES, pp 18–27

Niemann R, Marwedel P (1997) An algorithm for hardware/software partitioning using mixed integer linear programming. Des Autom Embed Syst 2(2):165–193

Nikolov H, Thompson M, Stefanov T, Pimentel AD, Polstra S, Bose R, Zissulescu C, Deprettere E (2008) Daedalus: toward composable multimedia MP-SoC design. In: Proceedings of the 45th annual design automation conference, DAC'08, pp 574–579

Padmanabhan S, Chen Y, Chamberlain RD (2011) Optimal design-space exploration of streaming applications. In: Proceedings of the IEEE international conference on application-specific systems, architectures and processors (ASAP), pp 227–230

Palesi M, Givargis T (2002) Multi-objective design space exploration using genetic algorithms. In: Proceedings of the international symposium on hardware/software codesign (CODES), pp 67–72

Panerati J, Sciuto D, Beltrame G (2017) Optimization strategies in design space exploration. In: Ha S, Teich J (eds) Handbook of hardware/software codesign. Springer, Dordrecht

Pimentel AD (2017) Exploring exploration: a tutorial introduction to embedded systems design space exploration. IEEE Des Test 34(1):77–90

Pimentel A (2020) A case for security-aware design-space exploration of embedded systems. J Low Power Electron Appl 10(3):1–12

Pimentel AD, van Stralen P (2017) Scenario-based design space exploration. In: Ha S, Teich J (eds) Handbook of hardware/software codesign. Springer, Dordrecht

Pimentel AD, Erbas C, Polstra S (2006) A systematic approach to exploring embedded system architectures at multiple abstraction levels. IEEE Trans Comput 55(2):99–112

Piscitelli R, Pimentel AD (2012a) Design space pruning through hybrid analysis in system-level design space exploration. In: Proceedings of the international conference on design, automation, and test in Europe (DATE), pp 781–786

Piscitelli R, Pimentel AD (2012b) Interleaving methods for hybrid system-level MPSoC design space exploration. In: Proceedings of the international conference on embedded computer systems (SAMOS), pp 7–14

Quan W, Pimentel AD (2014) Towards exploring vast MPSoC mapping design spaces using a bias-elitist evolutionary approach. In: Proceedings of the euromicro digital system design conference (DSD), pp 655–658

Quan W, Pimentel AD (2015) A hybrid task mapping algorithm for heterogeneous MPSoCs. ACM Trans Embed Comput Syst 14(1):1–25

Quan W, Pimentel AD (2016a) A hierarchical run-time adaptive resource allocation framework for large-scale MPSoC systems. Des Autom Embed Syst 20(4):311–339

Quan W, Pimentel AD (2016b) Scenario-based run-time adaptive MPSoC systems. J Syst Archit 62:12–23

Real E, Moore S, Selle A, Saxena S, Suematsu YL, Tan J, Le QV, Kurakin A (2017) Large-scale evolution of image classifiers. In: Proceedings of the 34th international conference on machine learning vol 70, pp 2902–2911. JMLR.org

Real E, Aggarwal A, Huang Y, Le QV (2019) Regularized evolution for image classifier architecture search. In: Proceedings of the AAAI conference on artificial intelligence, vol 33, pp 4780–4789

Reiss A, Stricker D (2012) Introducing a new benchmarked dataset for activity monitoring. In: 2012 16th international symposium on wearable computers. IEEE, pp 108–109

Sangiovanni-Vincentelli A, Martin G (2001) Platform-based design and software design methodology for embedded systems. IEEE Des Test Comput 18:23–33

Sapra D, Pimentel AD (2020a) Constrained evolutionary piecemeal training to design efficient neural networks. In: Proceedings of the 33rd international conference on industrial, engineering & other applications of applied intelligent systems (IEA/AIE 2020)

Sapra D, Pimentel AD (2020b) An evolutionary optimization algorithm for gradually saturating objective functions. In: Proceedings of the ACM international genetic and evolutionary computation conference (GECCO 2020)

Singh AK, Shafique M, Kumar A, Henkel J (2013) Mapping on multi/many-core systems: survey of current and emerging trends. In: Proceedings of the design automation conference (DAC), pp 1–10

Stefanov T, Pimentel AD, Nikolov H (2017) Daedalus: system-level design methodology for streaming multi-processor embedded systems-on-chip. In: Ha S, Teich J (eds) Handbook of hardware/software codesign. Springer, Dordrecht

Stierand I, Malipatlolla S, Fröschle S, Stühring A, Henkler S (2014) Integrating the security aspect into design space exploration of embedded systems. In: Proceedings of the IEEE international symposium on software reliability engineering workshops, pp 371–376

Tan B, Biglari-Abhari M, Salcic Z (2017) An automated security-aware approach for design of embedded systems on MPSoC. ACM Trans Embed Comput Syst 16(5s):1–20

Thompson M (2012) Tools and techniques for efficient system-level design space exploration. Ph.D. thesis, Universiteit van Amsterdam

Thompson M, Pimentel AD (2007) Towards multi-application workload modeling in sesame for system-level design space exploration. In: Vassiliadis S, Bereković M, Hämäläinen TD (eds) Embedded computer systems: architectures, modeling, and simulation. Springer, Berlin/Heidelberg, pp 222–232

Thompson M, Pimentel AD (2013) Exploiting domain knowledge in system-level MPSoC design space exploration. J Syst Archit 59(7):351–360

Thompson M, Pimentel AD, Polstra S, Erbas C (2006) A mixed-level co-simulation method for system-level design space exploration. In: Proceedings of the IEEE/ACM workshop on embedded systems for real-time multimedia (ESTIMedia'06), pp 27–32

Thompson M, Nikolov H, Stefanov T, Pimentel AD, Erbas C, Polstra S, Deprettere E (2007) A framework for rapid system-level exploration, synthesis and programming of multimedia MP-SoCs. In: CODES+ISSS'07: proceedings of the 5th IEEE/ACM international conference on hardware/software codesign and system synthesis, pp 9–14

Thoziyoor S, Ahn JH, Monchiero M, Brockman JB, Jouppi NP (2008) A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. In: Proceedings of the international symposium on computer architecture (ISCA), pp 51–62

Uhlig RA, Mudge TN (1997) Trace-driven memory simulation: a survey. ACM Comput Surv 29(2):128–170

van Stralen P, Pimentel AD (2010a) Scenario-based design space exploration of MPSoCs. In: Proceedings of IEEE international conference on computer design (ICCD), pp 305–312

van Stralen P, Pimentel AD (2010b) A trace-based scenario database for high-level simulation of multimedia MP-SoCs. In: Proceedings of the international conference on embedded computer systems: architectures, modeling and simulation (SAMOS), pp 11–19

van Stralen P, Pimentel AD (2012) A SAFE approach towards early design space exploration of fault-tolerant multimedia MPSoCs. In: Proceedings of international conference on hardware/software codesign and system synthesis (CODES+ISSS), pp 393–402

van Stralen P, Pimentel AD (2013) Fitness prediction techniques for scenario-based design space exploration. IEEE Trans Comput-Aided Des Integr Circuits Syst 32(8):1240–1253

Weichslgartner A, Wildermann S, Götzfried J, Freiling F, Glaundefined M, Teich J (2016) Design-time/run-time mapping of security-critical applications in heterogeneous MPSoCs. In: Proceedings of the 19th international workshop on software and compilers for embedded systems (SCOPES), pp 153–162

Wolf W, Jerraya AA, Martin G (2008) Multiprocessor system-on-chip (MPSoC) technology. IEEE Trans Comput-Aided Des Integr Circuits Syst 27(10):1701–1713

Xie L, Yuille A (2017) Genetic CNN. In: Proceedings of the IEEE international conference on computer vision, pp 1379–1388

Virtual Prototyping of Processor-Based Platforms

27

Tim Kogel

Contents

Introduction to Virtual Prototypes . . . 949
SoC Design and Verification Overview . . . 949
Historic Background of Virtual Prototyping . . . 951
Virtual Prototyping in the Verification Continuum . . . 952
Use-Cases for Virtual Prototypes . . . 954
Architecture Analysis . . . 956
Software Use-Cases . . . 960
Hybrid Use-Cases for Software-Driven Functional Verification . . . 963
System-Level Power Analysis . . . 965
Summary . . . 966
Building Transaction Level Virtual Prototypes . . . 967
The SystemC Transaction Level Modeling Standard . . . 967
Building TLM Components for Virtual Prototypes . . . 973
SSD Controller SoC Case Study . . . 977
SSD Controller SoC Introduction . . . 978
Loosely Timed Virtual Prototype of the SSD SoC . . . 979
Accurate Virtual Prototype of SSD SoC . . . 980
SSD Case Study Summary . . . 982
Conclusion and Outlook . . . 982
References . . . 984

Abstract

Virtual Prototypes of processor-based SoCs are widely used by semiconductor and system companies in many application domains like mobile communications, automotive, Internet of Things, storage, networking, and deep learning.

T. Kogel (✉)
Synopsys, Inc., Aachen, Germany
e-mail: tim.kogel@synopsys.com

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_44

They help to reduce risk, shorten time to market, increase design productivity, and improve the quality of results. This chapter motivates the usage of Virtual Prototypes and gives an extensive overview of the use-cases in the area of architecture analysis, software development, and functional verification of processor-based platforms. This chapter also covers the modeling of Virtual Prototypes, including an introduction of the SystemC Transaction Level Modeling (TLM) standard and the modeling of processors and peripheral components. In the last section, the described concepts of Virtual Prototyping are illustrated by an SSD Controller example.

Keywords

Electronic system level (ESL) · HW/SW co-design · Virtual prototype · Shift-left · Architecture exploration · Software bring-up · Software testing · Software-driven functional verification · SystemC · Transaction level modeling (TLM)

Acronyms

ADL	Architecture Description Language
AT	Approximately Timed
CA	Cycle Accurate
CPS	Cyber Physical System
CPU	Central Processing Unit
DLA	Deep Learning Accelerator
DMI	Direct Memory Interface
DSP	Digital Signal Processor
DUT	Device Under Test
ECU	Electronic Control Unit
EDA	Electronic Design Automation
FMI	Functional Mock-up Interface
FPGA	Field-Programmable Gate Array
FT	Fast Timed
FVP	Fixed Virtual Prototype
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HiL	Hardware in the Loop
HLS	High Level Synthesis
KPI	Key Performance Indicator
IP	Intellectual Property
IoT	Internet of Things
ISS	Instruction Set Simulator
LT	Loosely Timed
MAC	Multiply-Accumulate

MBD	Model Based Design
OS	Operating System
RTL	Register Transfer Level
SCML	SystemC Modeling Library
SLP	System Level Power
SoC	System-on-Chip
SW-VP	Virtual Prototype for software development
TLM	Transaction Level Modeling
UPF	Unified Power Format
UVM	Unified Verification Methodology
VP	Virtual Prototype
VPU	Virtual Processing Units

Introduction to Virtual Prototypes

This section first gives a brief overview of the design and verification of processor-based SoCs. Based on this overview, the need for Virtual Prototyping to cope with the growing complexity of architecture specification as well as software development and testing is discussed.

SoC Design and Verification Overview

Today, most chip designs are in fact System-on-Chips (SoC), integrating a mix of components such as several programmable cores (CPU, GPU, DSP, etc.), accelerators, different types of memories, a complex interconnect, and potentially other components (Greaves 2021). According to Handel Jones from IBS, "IC design costs have jumped from $51.3 million for a 28 nm planar device to $297.8 million for a 7 nm chip and $542.2 million for 5 nm" (Lapedus 2018). Curiously, the implementation of new hardware blocks is not a significant cost factor, mostly because the majority of components are re-usable IP blocks. Instead, much of the cost is attributed to software development and verification, but also architecture specification, IP qualification, and SoC-level hardware/software validation.

Figure 1 outlines the state-of-the-art SoC development flow. The classic hardware design entry is a Register Transfer Level (RTL) implementation based on a hardware Description Language (HDL) like Verilog or VHDL. This RTL description can be automatically taken to silicon using standard EDA tools for logic synthesis, place-and-route, etc. RTL is still a very detailed design entry, so it takes a high effort to implement and especially verify a hardware component at this level. The development of SoCs with multi-billion transistors is only feasible, because the design entry has moved to higher levels of abstraction. The most productive approach is to re-use available Intellectual Property (IP) blocks, which are developed internally or licensed from 3rd party IP providers.

However, custom design is still required for differentiated components, e.g., to accelerate a specific algorithm in order to achieve the necessary computational

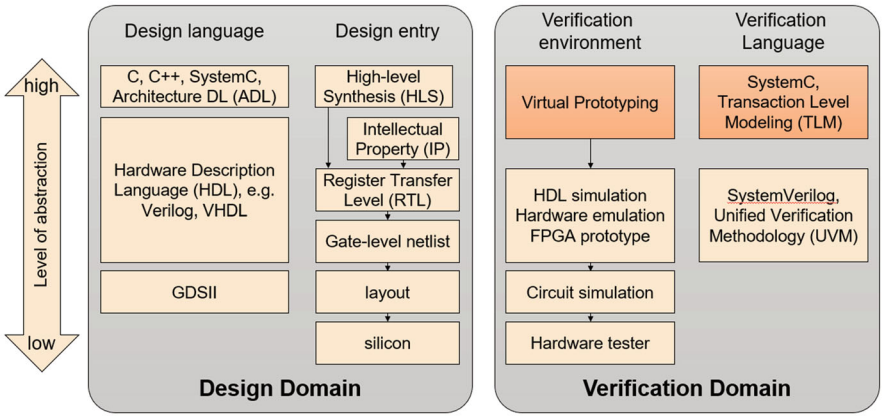


Fig. 1 SoC Design and verification

efficiency. In this case, High-Level Synthesis (HLS) is one option to improve productivity for the development of custom components by generating the RTL implementation from a high-level specification; see ▸ Chap. 24, "Accelerator Design with High-Level Synthesis" by Pilato et al. and on ▸ Chap. 28, "FPGA-Specific Compilers" by Zhang et al. There are generic behavioral synthesis tools supporting programming languages like C, C++, or SystemC, or more domain-specific tools using a specialized Architecture Description Language (ADL). The latter approach is very popular for the design of custom processors, because processor HLS tools also generate the required SW tools like compiler, assembler, linker, simulator, debugger, profiler, etc. (Hoffmann et al. 2001; ASIP Designer); see ▸ Chap. 25, "Processor Simulation and Characterization" by Martin et al.

On the verification side, the classic method is to verify the RTL description with directed tests written in the same hardware description language. Today, the recommended approach for functional hardware verification is based on SystemVerilog and the Unified Verification Methodology (UVM) (IEEE Standard for Universal Verification Methodology Language Reference Manual 2020). This supports state-of-the-art methods like constrained randomization, assertions, abstraction, etc. The slow speed of HDL simulation is addressed by hardware-assisted methods like emulation and FPGA prototyping to enable the verification and validation of large IP blocks and full SoCs.

The focus of this chapter is on the usage of Virtual Prototypes (VP) for processor-based platforms. The main use-cases are early architecture verification as well as debugging and testing of embedded software. The idea is to build an abstract transaction level simulation model of the SoC using the C++-based SystemC library. The next paragraph briefly summarizes the historic evolution before the pros and cons of Virtual Prototypes with respect to other verification methods are discussed in more detail.

Historic Background of Virtual Prototyping

Virtual Prototyping is in the tradition of HW/SW Co-Design, a term which was introduced in the 1990s to describe a new breed of design methodologies that would enable the transition from board-level to chip-level integration of processors, HW accelerators, interconnect fabrics, and memories (Edwards et al. 1997; Wolf 2003). Important concepts like system-level IP reuse (Martin 1998), interface-based design (Rowson and Sangiovanni-Vincentelli 1997), platform-based design (Keutzer et al. 2000), and virtual prototyping (Rowson 1997) were conceived in this decade. The three main branches of HW/SW Co-Design that emerged from this early work were system synthesis, architecture design, and HW/SW Co-simulation.

The ambition of early system synthesis frameworks was to reach the level of automation known from RTL logic synthesis, such that a formalized system specification is automatically partitioned and synthesized into optimized HW and SW implementation (Gupta and De Micheli 1993; Gajski et al. 1998). SoC-level system synthesis has not reached widespread industrial deployment, but the early work has bifurcated into Model Based Design for SW applications and High-Level Synthesis of HW components. Model Based Design (MBD) is widely used to generate a SW implementation from a high-level application models such as UML, SysML, or Matlab/Simulink (Friedman 2006; Liebel et al. 2018). As mentioned above, High-Level Synthesis of HW components is in active use for specific target architectures like FPGAs, application-specific processors, and accelerators.

The Polis approach (Balarin et al. 1997) pioneered the work on system-level frameworks for architecture modeling and design, which led to the first commercial offering called Virtual Component Co-Design (VCC). Today, commercial tools like Synopsys Platform Architect (Synopsys Platform Architect), Mirabilis VisualSim (Mirabilis VisualSim), and Intel CoFluent (Intel CoFluent) are available for user-directed modeling and exploration during the architecture specification phase.

HW/SW Co-simulation was pioneered by research on combining different models of computation (Buck et al. 1994). The practical deployment started out by connecting instruction set simulators to HDL simulators (Becker et al. 1992). This approach is still in use for functional HW verification, but the speed of HDL simulators is not sufficient to enable SW development use-cases. The simulation speed of the HW portion had to be improved by modeling at a higher level of abstraction. In the late 1990s, this was pioneered by different proprietary C-based languages like Magic-C from Virtio, CoWare-C from CoWare (Van Rompaey et al. 1996), VaST Virtual System Prototypes (Hellestrand 1999), and AXIS MaxSim (Guerra et al. 1999).

The proprietary nature of different modeling dialects limited the deployment of these early environments for HW/SW co-simulation. In 2000 the Open SystemC Initiative was formed to standardize on a language for system level modeling. In the same year, version 1.0 of the SystemC library was released, which was largely based on the Scenic design environment developed by the advanced technology group in Synopsys (Liao et al. 1997). SystemC 1.0 provided a C++ class library for modeling

of parallel HW modules and time, but in the initial version the communication was still limited to HW signals.

The next major release of SystemC 2.0 generalized the modeling of interfaces between modules, and this way enabled Transaction Level Modeling (TLM) (Groetker et al. 2002). Still, this did not ensure the level of interoperability required for the seamless exchange of models. Hence, the next major milestone was the standardization of a set of well-defined TLM interfaces in 2009. The SystemC library and TLM APIs were standardized by IEEE in 2005 and 2011, respectively (IEEE Standard for Standard SystemC Language Reference Manual 2012). As further explained in section "The SystemC Transaction Level Modeling Standard", the Loosely Timed and Approximately Timed APIs defined by the TLM standard are today the established interoperability standard for Virtual Prototypes.

Virtual Prototyping in the Verification Continuum

A multitude of tools and methods is required to cover the different aspects of SoC verification. Figure 2 qualitatively characterizes prominent verification methods, with respect to their suitability for architecture analysis and software development. In the context of this scope, the analysis is based on the following criteria:

- *Speed* refers to the time it takes to execute a hardware and/or software test.
- *Turn-around time* refers to the time to modify and rebuild the hardware and/or software implementation and/or test.
- *Early availability* describes at what time in the development process the respective method is available.
- *Timing Accuracy* denotes the level of timing detail and hence the suitability for performance analysis.
- *Debug and analysis* characterize the observability and controllability of the hardware and/or software under test.

	SDK, OS emulation	Virtual Prototype	HDL Simulation	Hardware Emulation	FPGA Prototype	Evaluation Board
Speed	++	+/-	--	0	+	++
Turn-around time	++	+	0	-	--	n/a
Early availability	+	+	0	-	--	n/a
Timing Accuracy	--	-/+	+	+	0	++
Debug and analysis	+	++	+	0	0	0
Accessibility & Collaboration	++	++	+	-	-	0

Fig. 2 Comparison of verification methods for architecture design and embedded software development. Rating: (+)+ → (very) high, 0 → intermediate, (−)− → (very) low

- *Accessibility* characterizes how easy it is for a developer to deploy a certain method. Typically simulation-based methods are more accessible, but cloud-based resource management solutions can help to improve the access of hardware-based methods.
- *Collaboration* characterizes how easy it is for individuals, teams, and companies to collaborate. Generally, methods based on abstract models are more suitable to enable collaboration without disclosing IP rights or confidential details.

Based on these criteria, we assess the suitability of different verification methods for architecture analysis and software development as shown in Fig. 2.

Software Development Kits (SDKs), like Android Studio (Android Studio), or *OS emulation* environments, like Silver for Autosar (Silver Virtual ECU), are very well suited for the development of embedded applications. However, due to high abstraction level of the underlying host-based OS simulation, they are not applicable for hardware-dependent software tasks like OS porting or driver development.

At the other end of the spectrum, software developers widely use *evaluation boards* to develop, debug, and test the hardware/software integration. The main disadvantage here is late availability, which puts the software bring-up into the critical path to bring the final product to market. Compared to simulation-based methods, real hardware has limited debug and analysis capabilities. Also, the dependency on hardware and associated lab equipment limits the accessibility and scalability of boards.

The sweet-spot of *HDL simulation* is block-level functional hardware verification. The slow simulation speed limits the applicability for architecture analysis and is prohibitive for software-related use-cases.

Hardware-based emulation mitigates the slow speed of HDL simulation, enabling SoC level performance measurements and software execution. However, the long turn-around time in the order of hours, and the dependency on stable RTL, limits the applicability for early architecture exploration.

FPGA prototypes have similar characteristics as hardware emulation, trading higher simulation speed for longer turn-around time. Also, in both methods the level of debug visibility is limited by the storage and/or external bandwidth requirements.

Finally, *Virtual Prototypes* close a gap in the capabilities of other methods:

- Virtual Prototypes for architecture analysis are relatively slow, typically in the range of 10–100 kcycles/s but provide the required level of timing detail for performance analysis. They are available early, ahead of RTL, and provide lots of analysis visibility to support the specification of the SoC specification with quantitative data. The fast turn-around time of abstract configurable models enables large-scale design-space exploration.
- Virtual Prototypes for software development trade less timing accuracy for much higher simulation speed, in the order of 10–100MIPS. The model-based approach enables software developers to shift left HW/SW integration debug and testing, but also to scale up to many users and software regressions testing.

In both cases, Virtual Prototypes enable the collaboration across teams and companies, e.g., system and semiconductor companies can jointly define the SoC architecture based on an executable specification, or semiconductor companies can enable their customers with a virtual HW/SW integration environment 6–12 months before silicon availability.

The total cost of deployment is of course another important criterion in addition to the technical aspects discussed above. Virtual Prototypes come with a price tag for tool and IP model licenses (or in-house development effort based on open-source software) as well as for the development of custom models and SoC model integration. The initial modeling effort for the first Virtual Prototyping project might be high, but typically many of the IP choices remain constant from one project to the next. Hence, the ROI greatly increases once a baseline library of models has been developed, and Virtual Prototyping is deployed as plan of record over multiple projects.

The remainder of this chapter is organized as follows: section "Use-Cases for Virtual Prototypes" introduces use-cases of Virtual Prototypes in more detail, including architecture specification and validation, early software development, software testing, software-driven functional verification, and early power analysis. After that, section "Building Transaction Level Virtual Prototypes" covers the modeling aspects, introducing the TLM 2.0 standard, the levels of abstraction to address the simulation speed and accuracy requirements of different use-cases, and in particular the modeling of processors and peripheral components. Finally, we illustrate the creation and usage of Virtual Prototypes based on a case-study from the SSD controller domain in section "SSD Controller SoC Case Study".

Use-Cases for Virtual Prototypes

The goal of this section is to describe different ways in which Virtual Prototypes are deployed and what kind of development problems they address. One important consideration is the specific requirements of the respective use-cases on the Virtual Prototypes in terms of simulation speed, functional and timing accuracy, visibility, flexibility, etc. These requirements drive the discussion of abstraction levels in the next section.

Figure 3 gives an overview of the major use-cases for Virtual Prototyping along the SoC product life-cycle and as they relate to semiconductor and system companies.

In the semiconductor development process, the implementation specification of a new SoC is an important milestone, which determines the major Key Performance Indicators (KPI) like performance, power, and cost. As described in section "Macro-architecture Specification" Virtual Prototypes for architecture analysis and exploration help to arrive at an optimal specification, such that the final SoC meets the KPIs for the target workloads.

Often the semiconductor vendor can only guess how the application workloads will look like, since this knowledge resides at the system companies building the actual end product around the chip. Here Virtual Prototypes for architecture analysis

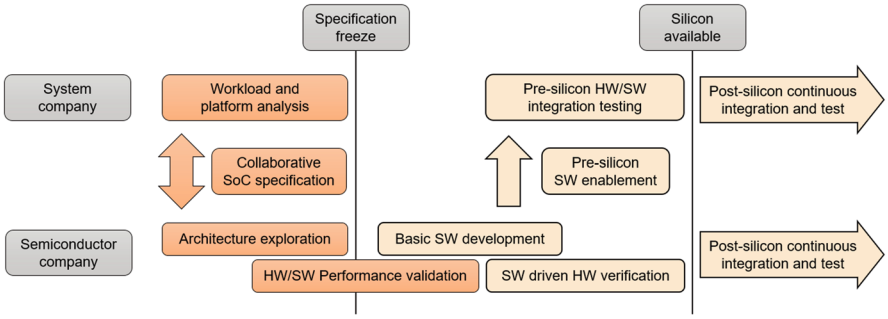


Fig. 3 Virtual Prototyping use-cases for early architecture analysis (left, dark orange) and software development (right, light orange) along the product life-cycle

enable early collaboration based on quantitative data between the semiconductor provider and user. The system company develops the workload models of their next generation products and evaluates SoC proposals from the chip suppliers. The Semiconductor vendor learns more about future use-cases and can tailor the SoC architecture to match these requirements.

After the specification freeze, Virtual Prototypes for software development (SW-VPs) enable the semiconductor vendor to "shift left" the development of hardware-dependent software, like OS porting and driver development. As described in section "Early Software Development", the basic SW together with the SW-VP can also enable the shift-left of the software development and testing tasks at the system company. Both for the semiconductor and system companies, the pre-silicon SW development and testing greatly shortens the time to market of the final product. After the silicon is available, the SW-VP continues to serve as a scalable target for continuous integration and testing (see section "Software Regression Testing").

Once the basic software is up and running on the SW-VP, it can also be used to drive more accurate HW/SW performance validation based on a more accurate Virtual Prototype (see section "HW/SW Performance Optimization and Validation"). As described in section "Hybrid Use-Cases for Software-Driven Functional Verification", the SW-VP can also be combined with RTL simulation, emulation, and FPGA prototyping for software-driven verification of the hardware under development.

In addition to the semiconductor and system related use-cases shown in Fig. 3, there is also an earlier stage usage by the IP providers developing Virtual Prototypes of reference designs to validate that their IP operates as expected in the SoC context. Publicly available examples are the Fixed Virtual Prototypes (FVPs) from Arm (Arm Fixed Virtual Platforms).

The following subsections present architecture analysis and software development related use-cases in more detail. Then follows a description of hybrid use-cases, which combine transaction level Virtual Prototypes with RTL models. The final subsection briefly introduces system level power analysis.

Architecture Analysis

Architecture analysis summarizes the use-cases for Virtual Prototypes related to the specification, exploration, optimization, and validation of SoC architectures. The focus is typically on non-functional properties like performance and power at different levels of scope, ranging from sub-system and SoC level to multi-chip level.

Figure 4 depicts the two main categories of architecture use-cases. In both cases, the goal is to assess the architecture with respect to the target KPIs, e.g., latency, throughput, average and peak power, energy per task, etc.

At first, these KPIs need to be estimated during the specification of the macro architecture, when high-level marketing requirements need to be translated into an implementation specification. The most productive approach for the early prediction of KPIs is based on rather abstract models of the architecture, as shown on the left side of Fig. 4.

Later in the development process, the KPIs need to be tracked and validated against initial assumptions and estimates. The validation of KPIs is performed using more detailed models of the architecture, as shown on the right side of Fig. 4. The following subsections discuss the use of Virtual Prototypes for architecture specification and validation in more detail.

Macro-architecture Specification

In this context, the macro-architecture refers to the configuration of an SoC sub-system, an SoC, or a multi-SoC design, i.e., the number and type of components and how they are configured and connected. In contrast, the micro-architecture is concerned with the implementation details of a single component, e.g., the processor pipeline or the structure of a Multiply-Accumulate (MAC) array inside a Deep Learning Accelerator (DLA). The specification of the macro-architecture has major impact on the non-functional properties of the final product in terms of performance, power, and cost.

Traditionally, the architecture specification and exploration is done with spread-sheets analysis, using static formulas to calculate the KPIs. For example, the

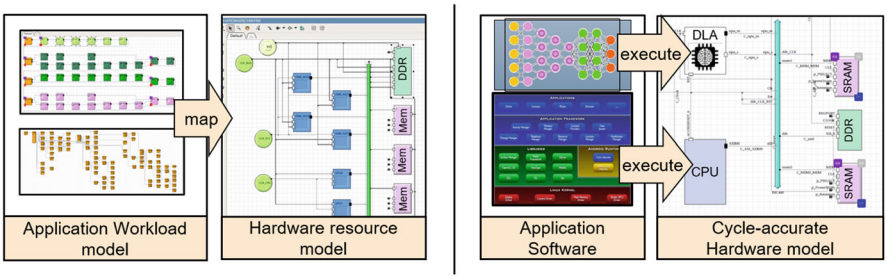


Fig. 4 Main architecture use-cases, left showing early architecture exploration and optimization with workload models, right showing performance validation with software

bandwidth requirements of all components in an SoC are accumulated to determine the necessary memory throughput and to dimension the interconnect and memory sub-system accordingly. Today, static spreadsheets are still helpful for analytical analysis of theoretical boundary conditions (Jünger et al. 2020). However, this approach does not allow the modeling of representative usage scenarios and the prediction of realistic KPIs for heterogeneous multi-processor platforms with complex cache and memory hierarchies running a mix of complex application stacks. As a result, specifications based only on static analysis tend to be either overly pessimistic, leading to over-design and excessive cost, or overly optimistic, leading to under-design and missed performance requirements. Both options result in schedule delays and/or less competitive products.

A suitable Virtual Prototype enables quantitative analysis of architecture options during the design specification phase. The goal is to complement static spreadsheets with a dynamic simulation model, which models the non-functional properties of the design in a more realistic way. A Virtual Prototype for this purpose needs to fulfill the following primary requirements:

- *Early availability* because hardware implementation and production software are typically not available during the architecture specification phase.
- Sufficiently *accurate timing* to measure performance related KPIs like latency, throughput, and utilization.
- High *flexibility and configurability* to explore the design space and determine the optimal selection, configuration, and connectivity of components.
- High *simulation speed and capacity* to execute a large number of scenarios. Typically, the problem can be easily parallelized into many simulations, each representing a different design configuration.
- *Model availability* for standard building blocks as they are available from 3rd party IP vendors.
- Very high *modeling productivity* for non-standard components and application scenarios, as the typical time window for architecture specification is 3–6 months, leaving little time for model development.

In general, the classic Y-chart approach (Kienhuis et al. 1997) is well suited to address these requirements. As illustrated on the left side of Fig. 4, the idea is to replace the missing software with a non-functional task graph, which does not model any behavior, but expresses the available thread-level parallelism and dependencies of the actual application as well as the processing and communication requirements. This task-based application workload model can be mapped to Virtual Processing Units (VPU), which model the execution resources, e.g., a CPU, GPU, DSP, or HW accelerator (Kempf et al. 2005; Kogel 2006). The VPU translates the communication requirements from the task graph into transactions, so the interconnect and memory resources can be represented as timing accurate TLM models. The timing is modeled by combining stochastic characterization of individual tasks and the simulation of dynamic effects like task scheduling, interconnect arbitration, and memory latencies.

This type of workload-based Virtual Prototype meets the requirements for macro-architecture specification. It is purely based on TLM models and hence does not depend on any software or hardware implementation. With the right modeling libraries, the Virtual Prototype is sufficiently easy to create, accurate, and fast. The flexible allocation of tasks to resources enables fast exploration of application mapping options. For commercial production projects, this modeling approach should be combined with the corresponding tooling and model library to achieve the necessary productivity for the creation, simulation, and analysis during the short architecture specification phase (Kogel 2017).

Under the umbrella of macro-architecture specification, a workload-based Virtual Prototype can be used for the following specific tasks:

- *HW/SW partitioning* by exploring the mapping of tasks to different number and types of processing resources, e.g., analyzing the impact of mapping a compute intensive task to a dedicated accelerator on the resulting cache performance and memory bandwidth. In the context of processor-based platforms, this includes the analysis of which processor types and how many processor cores are required, and how these cores need to be equipped with sufficient memory bandwidth.
- *Bus/memory optimization* by using the traffic generated by VPUs as stimuli for configurable TLM models of the interconnect and memory sub-system. Typical subjects of optimization are bus-width, clock frequencies, interconnect topology, number of outstanding transactions, QoS parameters, no-chip vs. off-chip memory, etc.
- *System level power analysis* by combining the workload-based performance model with a power analysis; see section "System-Level Power Analysis".

While a workload-based Virtual Prototype is a productive approach for early quantitative analysis during the architecture specification phase, there are also certain limitations. The fidelity of non-functional task graphs may not be sufficient to model data-dependent workloads, e.g., the activity of an SSD controller is highly dependent on the host command sequence.

Also, for certain applications it might be difficult to generate a highly accurate address sequence, e.g., as required for cache performance analysis. In general, the workload-based modeling approach requires data to characterize the task graph with the processing and communication requirements. This data might be available in IP data sheets or previous projects; otherwise the characterization needs to be based on initial estimates and assumptions, which need to be validated in subsequent design phases based on more accurate models.

Not all SoC design projects require macro-architecture analysis using workload-based Virtual Prototypes. Analytical architecture estimation based on spreadsheets is often sufficient for small and medium scale SoCs. On the other hand, the more complex the design, the higher the benefits of simulation-based architecture analysis. Also the stakes of taking wrong design decisions become higher for heterogeneous multi-processor platforms with complex cache and memory hierarchies running a mix of complex application stacks.

HW/SW Performance Optimization and Validation

The goal of performance validation is to measure KPIs using a more detailed representation of the system, addressing the limitations mentioned in the last paragraph of the previous section. For processor-based SoC platforms, this includes the execution of application software, or at least representative benchmark programs; see ▶ Chap. 25, "Processor Simulation and Characterization" by Martin et al. The traditional approach for accurate performance optimization and validation is using RTL-based methods. For complex sub-systems and full SoCs, this requires HW acceleration like emulation of FPGA prototyping. The late availability and long turn-around time for HW accelerated RTL-based methods motivates the usage of software-based timing accurate Virtual Prototypes for performance optimization and validation. Also, system companies typically do not license the IP themselves, so they do not have access to the RTL, and hence using timing accurate VPs is the only choice for pre-silicon performance analysis.

As depicted in the right side of Fig. 4, Virtual Prototypes for performance validation use fully functional cycle accurate models of all relevant components in a system. The programmable components need to be represented by Cycle Accurate (CA) Instruction Set Simulators (ISSes); see the discussion on cycle-level simulation in the ▶ Chap. 25, "Processor Simulation and Characterization" by Martin et al. Instruction Accurate ISSes are not useful for architecture analysis, since the software timing and especially the generated traffic are not accurate. Some processor IP providers offer ISSes with the capability to switch from fast instruction-accurate mode to cycle-accurate mode (Tensilica Customizable Processors).

Dedicated accelerators, like a Deep Learning Accelerator (DLA) executing a Neural Network, are similarly represented as fully functional timing accurate SystemC TLM models. For the interconnect and memory sub-system, the same set of timing accurate TLM models are used as in the workload-based approach. The Virtual Prototype needs to be sufficiently complete in order to execute the target software benchmarks. The timing of peripheral components can be abstracted in case they are not relevant for the performance of the system, like for a UART or an interrupt controller, etc.

The relatively high effort to build a cycle accurate Virtual Prototype and bring up the corresponding benchmark software impacts the flexibility and turn-around time for large-scale design space exploration. On the other hand, the high accuracy of actual software running on CA ISSes enables the following specific tasks:

- *Processor selection*: profiling of benchmark software on CA ISSes of the processor candidates allows accurate comparison of how the micro-architecture impacts the performance. For this purpose, the Virtual Prototype can be typically stripped down to the respective processor sub-system.
- *Cache optimization*: running the benchmark software on a CA ISS generates bus transactions with the accurate address sequence. Connecting the ISS to a configurable cache model enables optimization of cache parameters like cache size, number of associate sets, and replacement algorithm. In the context of

a cache-coherent interconnect, this also enables analysis and optimization of related parameters like coherency protocol, snoop filters, etc.
- *Bus/memory optimization*: similar to the workload-based approach, the traffic generated by the software running on the ISSes can be used as stimuli for configurable TLM models of the interconnect and memory sub-system to fine-tune parameters like buffer sizes and QoS regulators.
- *SoC architecture optimization for small and medium scale designs*: smaller devices are less affected by the limitations of cycle accurate Virtual Prototype. On the other hand, application domains like Internet of Things (IoT) or automotive micro-controllers benefit from the higher accuracy.

In general it is possible to mix workload-based and software-based methods in one Virtual Prototype, e.g., when certain sub-systems are re-used. In practice this is rarely done, since the slower CA ISS models limit the overall simulation speed and it can be difficult to mix data-less workload models with functional software execution.

This concludes the description of the main use-cases of Virtual Prototypes for architecture analysis. The following sections deal with use-cases related to software bring-up and testing.

Software Use-Cases

This section describes the usage of Virtual Prototypes for software-related use-cases like Operating System (OS) bring-up, driver debug, regression testing, etc. The common foundation is a transaction-level simulation model of the SoC capable of executing the unmodified software stack as it would also run on the real silicon. The emphasis is less on non-functional aspects like performance and power, but on highest simulation speed and sufficient functional completeness to cater to the requirements of software developers.

A typical flow for the creation and usage of Virtual Prototypes for software development (SW-VP) is shown in Fig. 5. A SW-VP contains all the components

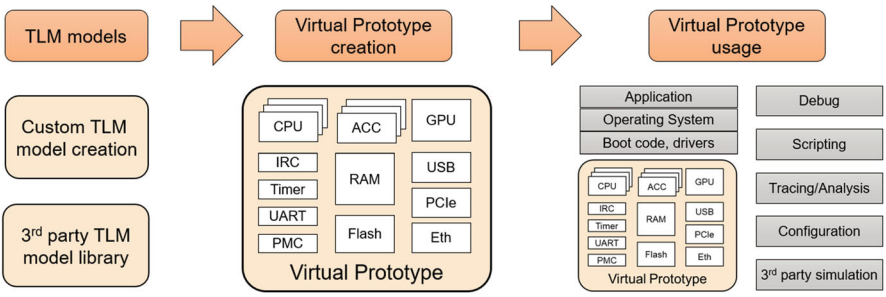


Fig. 5 Creation and usage of Virtual Prototypes for software development

that are visible to the programmers' view of the SoC, i.e., Instruction Set Simulators (ISSes) of the programmable cores, relevant peripherals like interrupt controller, timers, UART, memory, flash, etc. Depending on the needs of the SW stack that is executed on top of the SW-VP, further sub-systems like the power management controller, GPU, external IO, and hardware accelerators also need to be represented. The required transaction level models are either provided by the IP suppliers (Arm Fast Models; Synopsys DesignWare TLM library; Tensilica Customizable Processors; Intel Integrated Simulation Infrastructure with Modeling; CEVA), or need to be created by the SW-VP developer.

Independent of the specific use-case, SW-VPs provide several common benefits:

- *Non-intrusive and platform-level debug*: SW-VPs are integrated with standard debugger and SW IDEs, so the SW developer can debug the SW in the familiar tool environment. In addition, the virtual target provides additional debug visibility and controllability into all resources of the platform. Also, the virtual prototype can be observed and controlled in a non-intrusive way. This is particularly valuable for debugging multi-core platforms, where the SW on all cores can be simultaneously stopped and observed.
- *Scripting*: SW-VPs come with a scripting framework, which automates the SW execution and debug tasks, e.g., loading images, attaching scripts to software and hardware breakpoints, injecting faults to increase coverage, integration into regression frameworks, etc.
- *Tracing and Analysis* further improve debug productivity by providing intuitive visibility, e.g., through tracing of instructions, registers, functions, OS contexts, but also time-based logging of messages from the embedded SW and the simulation models.
- *Configuration*: the simulation model of the target can be easily reconfigured, e.g., extending the memory size to accommodate a larger debug build of the embedded SW.
- Integration with *3rd party simulators* enables users to model the system context of a SW-VP and simulate a more complete Cyber Physical Systems (CPSs) (Mueller et al. 2012).

Virtual Hardware in the Loop (HiL) is one important use-case in the automotive domain for the integration of Virtual Prototypes with 3rd party simulators (Feldner et al. 2019). As an example, a real-time engine controller application can be executed on the SW-VP of an Electronic Control Unit (ECU) (Reyes 2013). The external ECU interfaces are connected to a plant model of the engine, which can often be reused from a model-based design flow of the control algorithm, e.g., in Matlab/Simulink (Simulink). In this context, the Functional Mock-up Interface (FMI) (Functional Mock-up Interface) as defined by the Modelica Association (The Modelica Association) is an important interoperability standard for the integration of plant models into heterogeneous simulation environments.

The following sections describe the use-cases enabled by VPs for SW in more detail.

Early Software Development

As shown in the upper part of Fig. 6, the bring-up of embedded software on a custom SoC requires the silicon as an execution target. This puts the SW bring-up squarely into the critical path of bringing the complete product to market. Virtual Prototypes for early software development replace the silicon with a simulation model, so the SW bring-up can *shift left* and start while the SoC hardware is still being developed.

As an additional investment, the Virtual Prototypes for software development (SW-VP) need to be created. The creation of the VP should be split into multiple phases, which are aligned with the software development schedule:

- The OS bring-up requires only a rudimentary VP, containing the CPU, Interrupt Controller, Timer, and a UART.
- Further sub-systems need to be added to the VP in order to enable the development of the respective driver code, e.g., for PCIe, Ethernet, USB, etc.
- Developing and testing the secure boot code requires corresponding models of hardware security modules in the VP.
- Bring-up of Middleware layers, like graphics, audio, video, and AI, again requires the models of the corresponding sub-systems around the GPU, audio DSP, video/image processor, and AI accelerator.

This incremental approach enables the SW development tasks to start as early as possible and thus shortens the overall development schedule (Kang et al. 2019). Please refer to Section 3 of DeSchutter (2014) for a more detailed description on early bring-up of application software stacks.

Software Regression Testing

The bring-up of new code on a SW-VPs as described in the previous section is typically an interactive cycle of creating, compiling, and debugging the code. Once the new software is running in the local Sandbox of the developer, it needs to be verified against the bigger test suite to ensure that it does not break anything else in

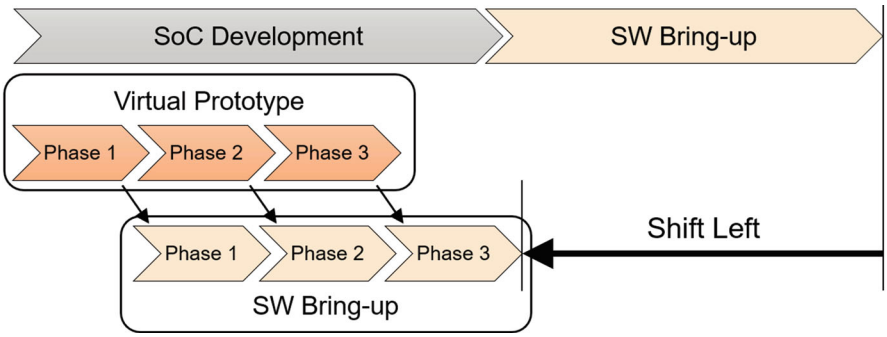


Fig. 6 Shift left concept, (top) traditional sequential bring-up process based on available hardware, (bottom) incremental parallel process supported by Virtual Prototypes

the system. The latter is typically a batch process referred to as *regression testing*, automated by a continuous integration server like Jenkins (Jenkins Automation Server for Continuous IntegrationContinuous Delivery). SW-VPs are well suited as an execution target for such a state-of-the-art DevOps flow for embedded SW, as they are more scalable, observable, manipulable, and deterministic as real HW (Bhote et al. 2019; Accelerate Devops 2019).

The typical regression flow is depicted in Fig. 7. The embedded software code is maintained in a revision control system for productive distributed development and version management. The same applies for the unit- and integration-tests as well as the Virtual Prototype, which both evolve over the product life cycle. Any modification triggers the continuous integration pipeline, including the build of the embedded SW and the SW-VP, and then the tests are executed in parallel on a compute farm. The scripting capabilities of the SW-VP allow the consolidation of the test results, comprising pass/fail and coverage reports as well as profiling of metrics such as SW execution times and memory footprint.

As indicated in the right side of Fig. 3, the usage of SW-VPs for regression testing extends into the post-silicon phase, well beyond the usage for early bring-up. Even when evaluation boards become available, the SW-VP is a more productive vehicle for regression testing, as they provide the scalability and availability to serve large and distributed SW development teams. The deterministic execution improves the robustness of the test flow, avoiding the unproductive hunt for false-negative test results due to non-deterministic hardware effects. Especially safety critical applications can benefit from the automated fault injection features of SW-VPs; see Oetjens et al. (2014) and Section 4.4 on Virtual Prototyping use-cases for embedded software testing in DeSchutter (2014).

Hybrid Use-Cases for Software-Driven Functional Verification

So far, we have considered "pure virtual" use-cases, where the Virtual Prototype is constructed entirely from TLM models. There are many good reasons for "hybrid" use-cases, where the SoC is split into a virtual TLM-based part and an RTL-based

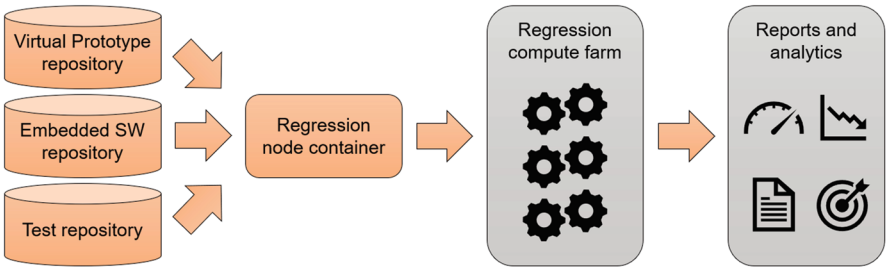


Fig. 7 Building a continuous integration pipeline for embedded software testing with Virtual Prototypes

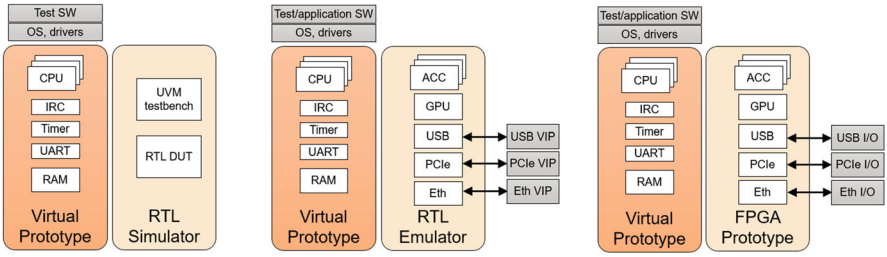


Fig. 8 Variations of Hybrid Prototyping use-cases: (left) RTL co-simulation, (middle) hybrid emulation, (right) hybrid FPGA prototyping

part. The typical setup is that the CPU sub-system executing the main SW stack is running on the virtual side, and varying portions of the remaining SoC are at the Register Transfer Level (RTL). This enables *software-driven functional verification*, checking the correctness of RTL in the context of the real SW stack.

As shown in Fig. 8, these hybrid use-cases are categorized depending on the execution environment of the RTL portion, into RTL co-simulation, hybrid emulation, and hybrid FPGA prototyping.

RTL Co-simulation

The primary use-case for RTL simulation is block-level functional verification, typically following the SystemVerilog based Universal Verification Methodology (UVM) (IEEE Standard for Universal Verification Methodology Language Reference Manual 2020). UVM advocates self-checking testbenches with constrained random stimuli generation for high coverage. To ensure that the Device Under Test (DUT) also operates as expected in the real-world context, directed SW-driven tests verify the RTL using actual SW drivers. Running the CPU executing the SW stack in the RTL would result in very slow simulation speed and long time to reach the actual start of the test after the OS is booted up. This simulation can be accelerated by running the SW on a virtual model of the CPU sub-system, especially when the TLM and RTL portions are simulated asynchronously (Mäenpää 2020).

Hybrid Emulation

Hardware Emulation can simulate a full SoC at RTL in the range of 5–10 MIPS, enabling the execute of real SW. However, the emulation of the CPU sub-system consumes large amounts of emulator resources, and booting an OS at typical emulation speed of 1–10 MIPS still takes in the order of hours. Again, replacing the CPU sub-system with a virtual model running at 100 MIPS saves emulator resources and accelerates the execution by 1–2 orders of magnitude, reducing the OS boot time to just minutes. With this cost and performance advantage, Hybrid Emulation has become the standard approach for SoC-level hardware/software verification.

In many cases, hybrid emulation is the entry point for using virtual methods. Commercial hybrid emulation solutions are available from all major EDA vendors, and the required virtual models of the CPU sub-system are provided by the IP

vendor (Arm Fast Models; DesignWare ARC nSIM). Compared to a full Virtual Prototype, the creation of TLM models of custom components is not required.

Hybrid FPGA Prototyping

FPGA Prototyping achieves even higher RTL execution speed than emulation, at the expense of a longer bring-up and turn-around time. The execution speed in the range of 30–50 MIPS enables the validation of the full SoC in the context of real-world IO. Hybrid FPGA Prototyping is not motivated by the relatively low incremental speedup of mapping the CPU sub-system to a virtual model. Instead, the main incentive is to shift-left the availability of FPGA Prototype of important peripherals such as PCIe, Ethernet, USB, HDMI, etc., which require validation with real-world interfaces. Waiting for the availability of the full FPGA Prototype would put the validation of key interface IP into the critical path.

Please refer to Chapter 6 in DeSchutter (2014) for a more detailed description and examples on Hybrid Prototyping and Hybrid Emulation.

System-Level Power Analysis

The focus of the previous sections is on using Virtual Prototypes for architecture analysis and for SW development. This section shows how to enable early power analysis as an overlay to the Virtual Prototyping use-cases discussed so far.

The annotation of power information is achieved by means of System Level Power (SLP) models, i.e., power models of IP components specifically for application in system level design use-cases. The format of system level IP power models is defined by the IEEE 1801-2015 standard (IEEE Standard for Design and Verification of Low-Power Integrated Circuits 2015). Originally, the IEEE 1801 Universal Power Format (UPF) was defined to capture power intent for hardware implementation and verification. UPF is a TCL (Tool Command Language, Tool CommLanguage) based format and defines the power supply and low power details as an overlay to the actual HDL implementation (Flynn et al. 2007). The System Level Power features of the 1801-2015 UPF 3.0 release extend the UPF TCL syntax to model power consumption as an overlay.

Figure 9 shows an example of a Virtual Prototype with a UPF 3.0 system level power overlay model. The power state machine defines the power consumption of

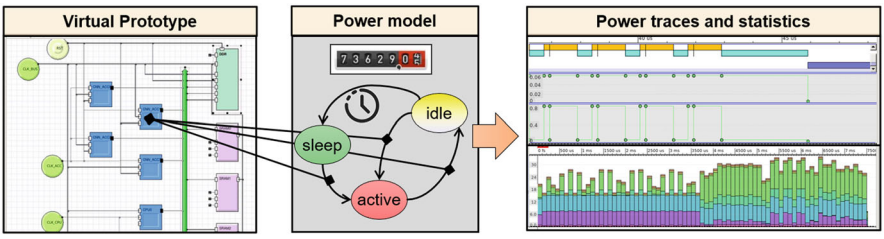


Fig. 9 Virtual Prototype with UPF 3.0 System Level Power Monitor

the respective component, either as a fixed value or as a configurable expression. The state transitions are triggered by observable events in the Virtual Prototype, e.g., whether a component is active, clock-gated, or in a low-power state. This way, the UPF 3.0 power model calculates and records the power consumption by observing the dynamic activity in the Virtual Prototype.

The goal of system level power analysis is not to provide 100% accurate power measurements, but to complement static power estimation based on spreadsheets with a more dynamic approach. The actual accuracy of system level power analysis depends mainly on the granularity of the power model and on the characterization of the power functions. The granularity of a SLP model is determined by the level of detail in the power model. For example, a CPU power model can be modeled as:

- a simple monolithic state machine as shown in the example above
- multiple domains for cores, cache, co-processor, etc., each with their own state machine
- a detailed instruction-level power model, which calculates power based on the specific energy of each executed instruction

The characterization determines how the power expressions calculate the power consumption based on power estimates or measurements.

- An early power characterization can be defined using high level estimates, e.g., based on the extrapolation of power measurements from previous projects.
- Once RTL and technology libraries are available, RTL or gate-level power estimation tools can be used to generate look-up tables, which determine power consumption based on design parameter configuration and operating mode.
- Post-silicon power measurements can still be valuable to characterize the power consumption of a re-usable IP block for usage in subsequent projects.

Despite the high level of abstraction, it turns out that Virtual Prototypes with system level power analysis models provide power estimates in the order of 85–90% accuracy, which are good enough to steer architecture design decision and to guide SW development in the right direction (Schürmans et al. 2013).

Summary

This section gave an overview of the many use-cases for Virtual Prototypes, covering early architecture analysis, early software development, regression testing, and RTL verification based on hybrid prototypes. The benefits in reducing risk, accelerating schedules, and increasing productivity of HW and SW design are driving an increasing deployment of Virtual Prototypes in many application domains (Arm Fixed Virtual Platforms).

- Early on, mobile application processors and modems were the driver for Virtual Prototypes to cope with the short design cycles (Aldis 2006).
- The complexity and accelerated innovation in automotive software and electronics is driving increased usage of VPs, especially to improve functional safety with virtual testing (Feldner et al. 2019).
- Many state-of-the-art SoC design projects in highly competitive application domains like Artificial Intelligence, SSD storage (Kang et al. 2019), SmartNIC, IoT, etc. deploy VPs for architecture design and pre-RTL SW development.

This deployment is supported by a healthy eco-system, providing models and tools for Virtual Prototyping, based on the interoperability enabled by the SystemC and TLM standard (IEEE Standard for Standard SystemC Language Reference Manual 2012). The next section discusses in more detail the creation of SystemC-based VPs for different use-cases.

Building Transaction Level Virtual Prototypes

A typical flow for the creation of Virtual Prototypes from Transaction Level Models is shown in Fig. 5. As discussed in previous sections, the suitable level of abstraction differs greatly depending on the target use-case of the VP. VPs for architecture analysis require timing accuracy, but can tolerate lower simulation speed, whereas VPs for SW development require highest possible simulation speed, but only minimal timing for the functional execution of the embedded SW. This section first defines more precisely the levels of abstraction for building Virtual Prototypes, starting with the modeling styles as defined by the SystemC TLM standard. After that the creation of TLM models for the different components in processor-based SoCs is presented.

The SystemC Transaction Level Modeling Standard

The IEEE 1666 standard for SystemC and Transaction Level Modeling (TLM) defines the widely accepted modeling language for the creation of Virtual Prototypes (IEEE Standard for Standard SystemC Language Reference Manual 2012). SystemC is a C++ library providing a set of classes to model system components and their communication interfaces, including a co-operative multitasking environment to model concurrent activity in a system.

On top of SystemC, the TLM library supports the abstract modeling of communication interfaces between SoC components. Since 2008 the TLM-2.0 standard provides a well-defined set of APIs and payload constructs to create interoperable TLM models for memory-map-based on-chip communication protocols. On the other hand, the TLM-2.0 standard is not really suitable for modeling serial interfaces for the integration of multiple SoCs. This is considered a missing capability,

especially for constructing Virtual Prototypes of bigger systems with multiple SoCs, like an automotive Electronic Control Unit (ECU) (Feldner et al. 2019).

At the synthesizable Register Transfer Level (RTL), a typical on-chip communication protocol like AMBA (AMBA AXI and ACE Protocol Specification 2013) is based on hundreds of individual signals for all attributes of a transaction, like address, data, control, and synchronization. The key idea of TLM is to model communication as set of function calls and a payload data structure representing the full semantics of the communication interface. This greatly reduces the number of synchronization points between communicating component models, which accordingly improves the overall speed of the event-driven SystemC simulation kernel.

To cover the modeling requirements of different VP use-cases, the IEEE Std 1666 TLM-2.0 Language Reference Manual (IEEE Standard for Standard SystemC Language Reference Manual 2012) identifies the following styles for the transaction-level modeling of memory-mapped on-chip bus protocols:

- The **Loosely Timed (LT)** modeling style aims to maximize the simulation speed by abstracting the communication to the highest level and by minimizing the synchronization overhead.
- The **Approximately Timed (AT)** modeling style focuses on the timing of the transactions between different components by providing multiple timing points for modeling individual transaction phases.

As illustrated in Fig. 10, both LT and AT modeling styles share the same concepts of sockets, generic payload, and an extension mechanism for modeling memory-

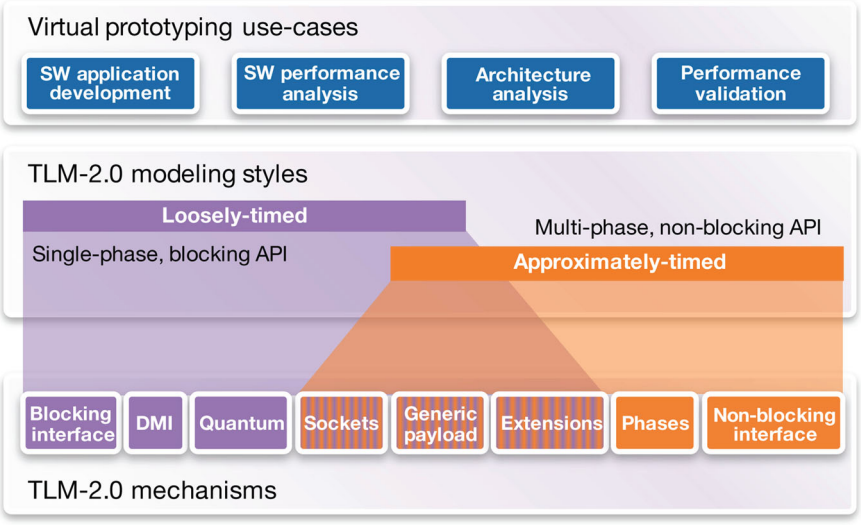


Fig. 10 TLM-2.0 modeling styles and mechanisms

mapped communication protocols. The extension mechanism allows adding custom attributes to the generic payload, which is important to model protocol specific attributes, like the transaction id, QoS, protection, cacheability, etc. in the AXI protocol (AMBA AXI and ACE Protocol Specification 2013). This common infrastructure enables the smooth integration of models using different modeling styles. On the other hand, LT and AT leverage specific mechanisms to cater to the different requirements of different Virtual Prototyping use-cases for software development and architecture analysis.

Loosely Timed Modeling Style

Virtual Prototypes for software development are created to provide an abstract model of the target hardware platform, which can execute the unmodified software. The key requirements are:

- Register accuracy: In order to run embedded software correctly, the memory and memory-mapped register layout and behavior of all relevant components should be modeled.
- Functional fidelity: All relevant responses of the target hardware should be modeled.
- Simulation speed: It is important that the Virtual Prototype can execute the embedded software at a speed that is as close as possible to real time of the actual target device.

The Loosely Timed (LT) modeling style is intended to maximize the execution speed while providing the minimal level of required timing fidelity. The key concepts in the TLM-2.0 standard to achieve high simulation speed on top of the event-driven SystemC simulation kernel are temporal decoupling, the Direct Memory Interface (DMI), and blocking communication.

- Temporal decoupling allows initiator components, like processor models, to run ahead of the global time for a maximum quantum of time before synchronizing with the SystemC kernel; see Engblom (2018).
- DMI allows initiator components to bypass the regular TLM interface and directly access instruction and data memory via the simulation host address.
- The simple blocking TLM transport interface is used for non-DMI accesses to memories and peripheral registers. This is required for accesses that trigger a specific behavior at the target side, like a register access to a DMA controller that actually triggers the DMA transfer. As depicted on the left side of Fig. 11, LT communication is modeled using a single function call.

The LT modeling style reflects the software interactions with hardware and how register content is updated. For example, timer interrupts happen roughly at the intended time to simulate the timing calibration loop in a Linux boot, or to execute real-time software in automotive Electronic Control Units (ECUs).

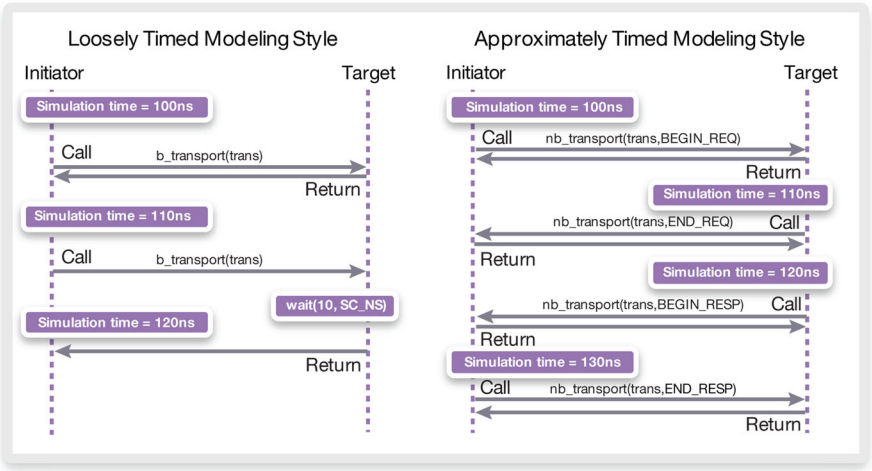


Fig. 11 TLM-2.0 Protocols for Loosely Timed (left) and Approximately Timed (right)

Extended Loosely Timed Modeling Style

The TLM-2.0 generic payload only covers the common subset of transaction attributes like address, data, and length. The extension mechanism allows to include additional protocol specific attributes to the generic payload, e.g., security extensions, atomic transactions, coherency flags, etc. Based on this extension mechanism, owners of on-chip bus protocols can define a layer on top of TLM-2.0 for creating protocol specific models in an interoperable way. For example, Arm provides a library of "AMBA-PV Extensions to TLM," which enables modeling of AMBA buses using an LT coding style (Arm Ltd 2020).

The Loosely Timed modeling style has been very successful in fostering the availability of interoperable models from all major IP providers (Arm Fast Models; Synopsys DesignWare TLM library; Tensilica Customizable Processors). The availability of LT TLM models for off-the-shelf IP blocks has significantly reduced the effort to create Virtual Prototypes for SW development.

Approximately Timed Modeling Style

Virtual Prototypes for early architecture analysis and exploration are created in order to provide an abstract model of the target hardware, which reflects relevant performance metrics like bandwidth, throughput, utilization, and contention. The key requirements are:

- Scalable timing accuracy: The accuracy requirements depend on the goal of the project. For example, an abstract model of a DRAM is good enough for exploring HW/SW partitioning, but a highly accurate model is needed for optimizing the configuration of the DRAM memory controller.

- Compositional timing: the end-to-end performance of a system can be obtained from assembling a set of components which only model their individual timing.

Compared to the LT modeling style described previously, the Approximately Timed (AT) modeling style is intended to model the communication with more detailed timing. As shown on the right side of Fig. 11, a single transaction is broken into multiple phases to reflect the timing of a bus protocol in more detail. The non-blocking TLM transport interface is used to mark start and end of each phase.

The TLM-2.0 initiator and target sockets bundle a forward and backward path into one interface to enable bi-directional communication. An initiator calls nb_transport via the forward path to mark the begin of a request phase. The target can decide to finish the request phase immediately, potentially with a delay annotation. Alternatively, the target can call nb_transport some time later via the backwards path to explicitly mark the end of a request phase. As depicted in the right side of Fig. 11, the TLM-2.0 standard defines an AT "Base Protocol" (AT-BP) with a request and response phase marked by four distinct timing points. This enables the modeling of basic communication aspects like throughput, latency, and transaction pipelining.

Extended AT

The TLM2.0 AT base protocol has limited expressiveness when it comes to accurately representing real-world on-chip bus protocols:

- It does not provide timing points for the individual data beats of a burst transfer. This becomes particularly problematic when interfacing TLM-2.0 AT-BP with cycle accurate or RTL models.
- It requires all address and data information to be available for writes at the start of the transaction. Equally, all data for a read response needs to be available at the start of the response.
- It is not possible to have concurrent read and write requests, as, e.g., required by the AMBA AXI protocol (AMBA AXI and ACE Protocol Specification 2013).

To overcome these deficiencies of the AT base protocol, the TLM-2.0 standard provides an extension mechanism for the AT modeling style, which enables the definition of additional protocol phases and timing points. Together with the extension mechanism for the generic payload, which is also used for Loosely Timed modeling, this enables the more accurate modeling of on-chip bus protocols. In fact, the AT extension mechanism allows the definition of fully cycle accurate modeling of real-world bus protocols.

The Fast Timed (FT) modeling style that is part of the SystemC Modeling Library (SCML) (The SystemC Modeling Library) from Synopsys is an example of using the TLM-2.0 AT extensions to create more accurate TLM protocols. The idea is to define an extended set of protocol agnostic phases, like RD_ADDR_START/_END, RD_DATA_START/_END, etc., that can be mapped to specific protocols, like AMBA AXI. As shown in Fig. 12, this enables the accurate modeling of all phases of a parallel read and write transaction on an AXI bus.

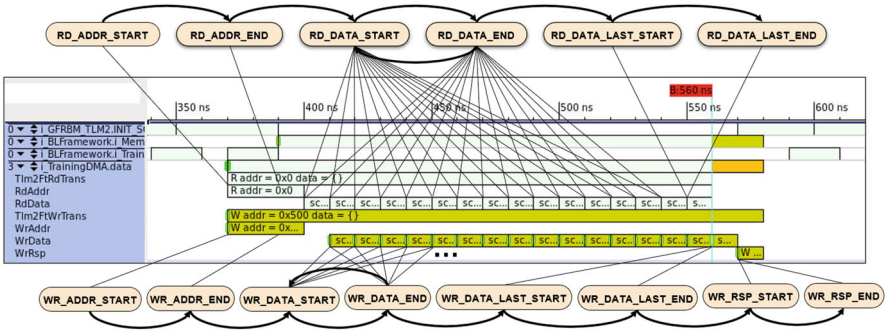


Fig. 12 TLM-2.0 Extended Fast Timed Protocol example representing AMBA AXI read (top) and AXI write (bottom)

Protocol specific extensions can break the interoperability between the TLM-2.0 AT Base Protocol (AT-BP) and extended AT models. The FT modeling style is defined using ignorable extensions in a way that enables the definition of more accurate protocols while preserving interoperability with the AT-BP.

- FT extends the generic payload with an optional attribute indicating the current state in the protocol state machine.
- FT uses extended sockets that provide the necessary protocol conversion logic such that conversions are only done when required and can be inserted automatically. This way the extended set of FT protocol phases can be mapped to the standard set of AT-BP phases.
- For each specific protocol, protocol-specific attributes are added as needed, e.g., for cacheability, out-of-order transactions, etc. This should be limited to those attributes that are not already covered by the TLM2.0 base protocol, so the extended protocol can fall back to the more generic AT base protocol.

Specific FT protocols represent the full set of protocol phases and transaction attributes of the original on-chip bus protocol. This enables the creation of fully cycle accurate FT models, where the accuracy is not limited by the expressiveness of the TLM protocol, even though certain details of the model-internal behavior might be abstracted. Accordingly, fully cycle accurate transactors between TLM models and pin-level RTL models can be created, which map the TLM FT protocol phases to RTL pin events and vice-versa. This way, implementation accurate RTL models can be used in the context of accurate TLM models.

TLM-2.0 Summary

The goal of the IEEE 1666 TLM-2.0 standard is to enable model interoperability at the level of SoC building blocks, like processors, buses, memories, or peripherals. For this purpose, TLM-2.0 standardizes the modeling interface for memory mapped

bus communication, which is the prevalent SoC interconnect mechanism. The LT and AT modeling styles cater to the different requirements of different use-cases like SW development and architecture analysis. Although AT allows more detailed timing modeling than LT, the modeling style of the communication interface should not be confused with the abstraction level or timing accuracy of a model itself. LT and AT only refer to the communication aspect, whereas abstraction and timing accuracy also depend on the timing and granularity of the structure and behavior inside the component.

Building TLM Components for Virtual Prototypes

After describing the interfaces between the components of a Virtual Prototype, this section describes the modeling of the components themselves. This part is not governed by any standards, as long as the interface adheres to the standard TLM API. Still, a common set of guidelines and recommendations can be identified for the modeling of processors, accelerators, and other peripheral components.

Developing TLM models is the biggest investment to enable the deployment of Virtual Prototypes. However, with an IP-based SoC design paradigm, the majority of components are reused, either by licensing 3rd party IP, or by reusing in-house IP across many SoC projects. In the same way, the TLM models are provided by the IP vendor or by the in-house IP development team, such that the investment into the development of TLM models can be amortized over many projects. On the other hand, many SoCs also include unique IPs to differentiate the SoC for a certain application domain, e.g., a customized processor or accelerator. Suitable TLM models need to be developed for these custom components in order to enable Virtual Prototyping.

This section gives some guidance for the development of TLM models, starting with some general considerations about abstraction levels and discussing the development of processor and peripheral models in more detail.

Levels of Abstraction

Obviously, it would be preferable to have one single model that enables all VP use-cases. Unfortunately, the *vicious modeling triangle* proclaims that any model can only fulfill two of the three desirable attributes of being fast, accurate, and economical to develop.

- LT models are fast and their development requires relatively low effort, but they do not provide sufficient timing accuracy for architecture use-cases.
- RTL simulation or translation is accurate and cost-effective, but the simulation speed is not sufficient to enable architecture exploration or SW development related VP use-cases.
- Hand-written speed-optimized timing-accurate TLM models or RTL emulation are both fast and accurate, but also rather expensive solutions.

One escape route from this vicious triangle is to generate fast timing accurate models from a higher level formalized specification, but developing such technology requires high initial investment and is only applicable to a constrained class of target IP (Hoffmann et al. 2001; ASIP Designer; Tensilica Customizable Processors; Lecler and Baillieu 2011).

Commercial TLM libraries abandon the goal of providing a one-fits-all solution and typically focus on the more widely deployed SW related use-cases (Arm Fast Models; Synopsys DesignWare TLM library). Vendors of interconnect and memory controller IP that is critical for the SoC performance focus on providing Approximate Timed models for architecture analysis (Lecler and Baillieu 2011; Synopsys Platform Architect). Embedded processor vendors often provide two versions of models for their IP, one for SW use-cases and one for architecture use-cases, e.g., Arm's Fast Models (Arm Fast Models) and Cycle Models (Arm Cycle Models), Synopsys Arc nSim and xCAM (DesignWare ARC nSIM), Tensilica TurboXim, and cycle accurate ISS (Tensilica Customizable Processors). Open-source processor modeling frameworks like gem5 provide multiple levels of abstraction for the processor core and the memory sub-system to choose the most suitable speed-accuracy trade-off for the respective modeling task (Binkert et al. 2011).

Processor Models

Programmable cores are obviously the key component of processor-based platforms, and therefore the corresponding processor models play a critical role for the Virtual Prototype. The technology used for the development of the processor models largely determines the simulation speed and accuracy of the overall VP.

- *Host-based Simulation* (HbS) is used for highly abstract models targeting application development, e.g., for mobile apps (Android Studio) or in virtual ECUs for automotive applications (Silver Virtual ECU). The details of the target hardware are abstracted, and the application software is executed on the simulation host.
- An *Instruction Accurate Instruction Set Simulator* (IA-ISS) executes the target code of the embedded software without modeling the details of the micro-architecture, like processor pipeline, instruction pre-fetching, branch prediction, instruction level parallelism, etc. Advanced simulation techniques like dynamic binary translation (Nohl et al. 2002; The Software Freedom Conservancy) are used to maximize the simulation speed. IA-ISSes can be combined with timing annotation to estimate cycle counts of SW benchmarks.
- A *Cycle-based ISS* models the processor timing with higher timing fidelity at the expense of lower simulation speed. The manual effort for the creation, validation, and maintenance is very high, but there are tools that can automatically generate a cycle-based ISS from an architecture description of the processor (Hoffmann et al. 2001; ASIP Designer; Tensilica Customizable Processors).
- *RTL* co-simulation, translation, or emulation are alternative sources of accurate processor models, which can be used when TLM models are not available and for use cases that require highest accuracy (Arm Cycle Model Studio; Verilator).

Please refer also to the ▸ Chap. 25, "Processor Simulation and Characterization" by Martin et al. for a more detailed description of the different abstraction levels of processor simulators.

TLM Integration of Processor Models

The processor models described in the previous paragraph need to be integrated into a SystemC-based TLM environment in order to deploy them in the context of a Virtual Prototype. This integration effort comprises many facets, like choice of interfaces, granularity, debugger integration, analysis instrumentation, etc., which determine the usability of the final Virtual Prototype.

The choice of the appropriate TLM interface typically follows the abstraction level of the processor model.

- An Instruction Accurate ISS naturally fits with the Loosely Timed blocking transport API and Direct Memory Interface. Also, the simulation loop of the processor can be smoothly integrated with the temporal decoupling concept of the LT API to achieve the highest possible simulation speed and still maintain sufficient timing fidelity in the context of a multi-core VP (Engblom 2018).
- A Cycle-based ISS as well as RTL cores should be fitted with the non-blocking transport interface of the Approximately Timed TLM-2.0 API, preferably an extended version that enables the modeling of individual data phases. This way the inherent accuracy of the processor model is preserved in the context of the VP, such that the generated TLM traffic can be used for architecture use-cases like performance analysis of caches, cache coherent interconnect, Network-on-Chip and memory controllers, etc., all of which benefit from highly accurate transactions sequences.

Another important aspect of the integration is the right granularity of the integration. The minimum granularity is the processor core, as it is represented by the ISS. Typically, an entire processor cluster including level one and sometimes even level 2 caches is readily integrated into a top-level module, as it simplifies the deployment of the model in the context of a bigger VP. The important parameters of the processor cluster like number of cores, cache sizes, etc. should be configurable.

Additional features increase the usability of the integrated processor model in the context of a VP.

- *Processor debugger support* is critical to enable software development use-cases. For heterogeneous multi-core platforms, a sophisticated debug server infrastructure is required to coordinate multiple debugger connections.
- *Image loading* is a convenient feature where the processor model automatically loads the executable into the platform memory, in the right format and at the right location.
- *SW-centric platform visibility* can be enabled by leveraging the TLM-2.0 transport debug API to visualize and manipulate the content of memories and peripheral registers.

- *Tracing, analysis, and scripting* provide further added value for all VP use-cases. The processor model needs to provide the necessary APIs to peek and poke into local resources like registers, memories, performance counters, etc. Based on these instrumentation APIs, Virtual Prototyping tools can visualize traces of instructions, function calls, and register access as well as OS-aware tracing of context switches.

In summary, the integration of processor models into a Virtual Prototyping environment is a development task in its own right, especially providing a full-featured Virtual Prototyping experience with all the value-added features.

TLM Models of Peripheral Components

Thanks to the TLM-2.0 interoperability standard, today many TLM-2.0 compliant models of standard SoC components like processors, buses, and memories are available from the respective IP provider. However, there are still a significant number of custom building blocks like timers, interrupt controllers, DMA controllers, or hardware accelerators, for which specific models need to be created. TLM-2.0 defines the interoperability standard, but it does not prescribe how to model the internal behavior. In order to reduce the actual modeling effort, a well-defined modeling methodology and a library of re-usable modeling objects is required. In larger companies this is especially important to unify the modeling style across distributed modeling teams.

This section explains the concept of modeling objects and patterns based on the publicly available SystemC Modeling Library (SCML) from Synopsys (The SystemC Modeling Library).

SCML is a layer on top of SystemC and TLM-2.0. It hides a lot of the complexity and common code that is required to correctly manage TLM2 transactions, and it provides with modeling objects that handle common aspects of Virtual Prototype modeling. The modeling objects in the SCML promote the separation of communication, behavior, and timing (Kogel 2006). This way the models created based on this methodology support different modeling styles like LT, AT, and FT.

Figure 13 illustrates the coding style, which is enabled by the SCML modeling objects. The key idea is to separate the actual behavior of the component from the external bus interface of the interconnect model. This separation between the extended interconnect protocol and the generic TLM2.0 protocol used by the SCML storage objects is implemented by a protocol adaptation layer. The actual behavior of the component can be separated into a storage and synchronization layer and the pure functional behavior of the model:

- The storage and synchronization layer stores the data of write transactions and returns the data in case of read transactions.
- The behavior models the algorithm or state machine of the component. The behavior is triggered when certain memories or registers in the storage layer are accessed.

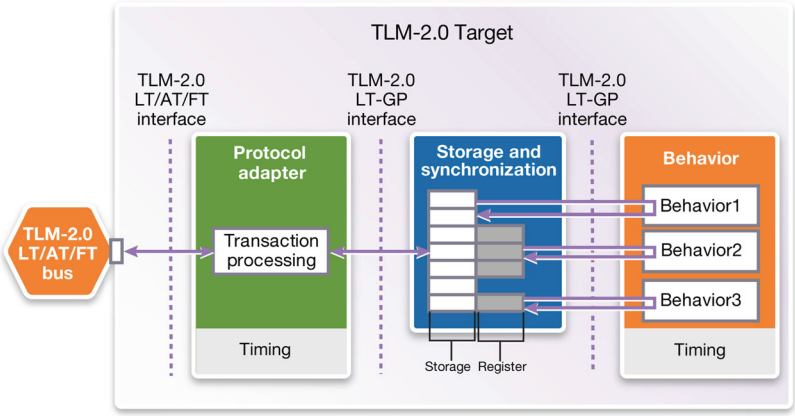


Fig. 13 SCML-based modeling pattern for target peripherals

The different needs in timing accuracy can be addressed by separating the code that models the timing of the component from the pure functional behavior. SCML supports this separation by providing modeling objects for each of these layers.

- The adaptation layer handles communication related and data-independent timing aspects, like the duration of a protocol phase or the number of outstanding transactions.
- The behavior layer handles processing related and data-dependent timing aspects.

The SCML modeling library greatly helps to reduce the modeling effort of custom peripheral components. In the context of commercial Virtual Prototyping projects, the effort for creating models can be further reduced by using model generation tools. For example, an SCML-based peripheral model can be automatically generated from an IP-XACT description of the register interface.

SSD Controller SoC Case Study

The last section of this chapter illustrates the construction and usage of Virtual Prototypes based on an SSD controller SoC case study. Especially in the area of data center storage, this is a representative example of a demanding and competitive application domain, both in terms of complex multi-core SoC architecture and complex software stacks in the critical path of the system performance. The following sub-sections give a brief introduction to SSD controllers and then describe the refinement from a Loosely Timed Virtual Prototype for SW development to timing accurate Virtual Prototype for performance analysis.

SSD Controller SoC Introduction

A typical SSD system is shown in Fig. 14. A more in-depth background on SSD memories and controllers is provided by Micheloni et al. (2018) and the OpenSSD project (The OpenSSD Project). The SSD controller translates NVMe host commands into NAND Flash operations. This involves several processing tasks to mitigate the aging of the NAND Flash devices with every program/erase cycle. The central function in the SSD firmware is the Flash Translation Layer (FTL), which distributes host requests evenly among the physical NAND resources. For this purpose, the FTL maintains a map to translate logical host addresses into physical Flash addresses. A typical sequence of FTL operations for reading and writing to the Flash device is shown on the left side of Fig. 14. The complexity comes from the very high throughput requirements for data center SSD controllers, where the FTL operations are in the data path and need to be performed with the highest possible performance.

The typical components in an SSD controller SoC are shown on the right side of Fig. 14.

- The central component is a high-performance multi-core real-time CPU, which can efficiently process the high frequency of interrupts from the other components.
- A DDR memory is required to buffer incoming host requests and maintain the FTL tables.
- DMA engines offload the CPU from moving the payload between host interface, DDR memory, and flash controller.
- The flash controllers manage the low-level operations to access the actual NAND devices.
- The interface to the host is a high-throughput NVMe controller with a PCIe physical layer.

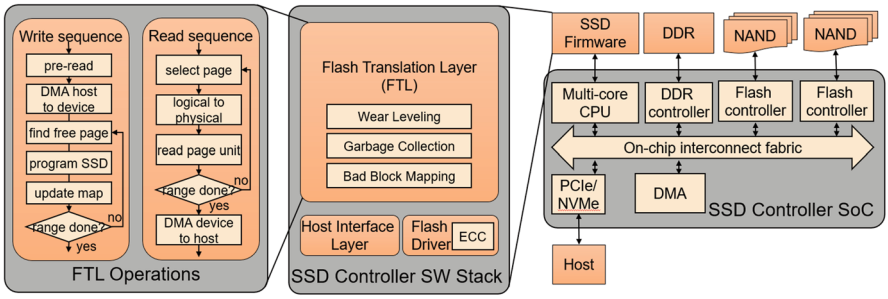


Fig. 14 Typical SSD system with SoC block diagram (right), SSD firmware components (middle), write and read operation (left)

Optimizing the performance for such a complex application is a phase-coupled problem: the HW architecture needs to be optimized for the SW application and vice versa. We break this mutual dependency by starting with a fast Loosely Timed Virtual Prototype for software development (SW-VP) to bring up an initial version of the SSD firmware stack. Then the SW-VP is successively refined into a timing accurate VP for performance optimization by replacing the timing critical components with their accurate counterparts (Kang et al. 2019).

Loosely Timed Virtual Prototype of the SSD SoC

As shown on the bottom right side of Fig. 15, the first step is to build a fast Loosely Timed Virtual Prototype, which enables the execution of the unmodified SSD firmware stack. This example is based on an Arm CPU, so the CPU subsystem is modeled using an Arm Fast Model of the Cortex R processor and the Generic Interrupt Controller (GIC) (Arm Fast Models). The host interface is based on the Synopsys PCIe controller, which is available in the TLM model library of Synopsys Virtualizer, a Virtual Prototyping environment for software development use-cases (Synopsys Virtualizer; Synopsys DesignWare TLM library). The same applies to the Loosely Timed models of the generic NVMe controller, interconnect, memory, DMA controller, and UART. The SSD firmware in this case-study is based on the OpenSSD project, see Song et al. (2014) and The OpenSSD Project.

Only the Flash Controller is a custom IP block and therefore requires a dedicated modeling effort. Following the modeling pattern based on the SCML modeling library (The SystemC Modeling Library) as described in the last paragraph of section "Building TLM Components for Virtual Prototypes", the actual coding is reduced to the functional behavior. The modeling effort is further reduced by using the Virtualizer TLM Creator tool with its library of reusable TLM building

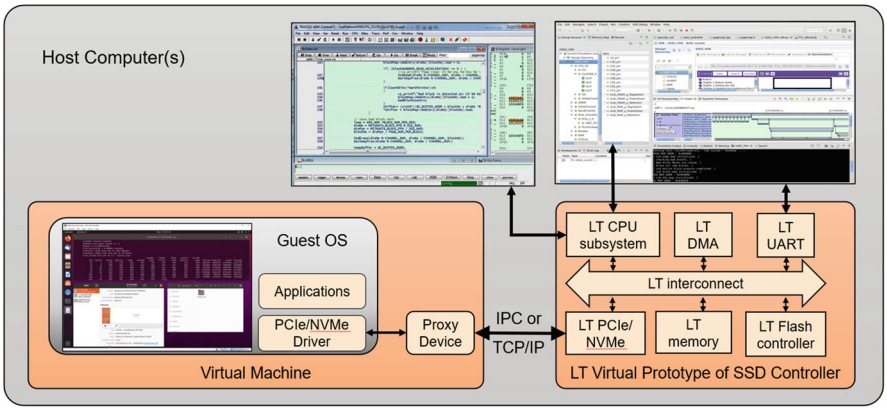


Fig. 15 Loosely Timed Virtual Prototype of SSD controller, connected to embedded software debugger, platform debugger, and OS for end-to-end software testing

blocks for basic components like FIFO buffers, state machines, and the ONFI Flash protocol interface. Apart from the required functionality, the flash controller model is annotated with configurable timing to take the latency of Flash operations into account. Also, a variety of custom monitors are added to trace and analyze, e.g., the IO operations per second, the flash commands and internal state, and the accessed flash blocks. Enriching the flash controller model with timing annotation and analysis instrumentation requires some additional modeling effort, but greatly increases the usability for SW development and architecture analysis use-cases.

By itself, the Loosely Timed Virtual Prototype of the SSD controller can be used to test the embedded firmware. The real value and practical usability come from the integration of the SW-VP into an ecosystem of software development tools:

- A software debugger, like gdb, Arm Development Studio (Arm Development Studio), or Lauterbach Trace 32 (Lauterbach Development Tools), can be attached to the ISS in order to debug the embedded SSD firmware.
- The Virtualizer Studio platform debugger can be attached to the VP to show traces and analysis views, e.g., peripheral register trace, memory content, multi-core function, and OS traces, and custom views like ONFI trace, etc.
- With a special proxy device, the TLM model of the Synopsys DesignWare PCIe End Point can be connected to the Root Complex in a Virtual Machine like QEMU (The Software Freedom Conservancy) or Oracle Virtual Box (Oracle Virtual Box).

The PCIe Virtual IO connectivity enables the testing of the firmware on the SSD device in the context of the host driver and host application. This way, the VP of the SSD controller can be mounted, enumerated, formatted using standard Linux commands, and accessed from applications like file browsers or benchmark programs, just like a real NVMe device. This end-to-end testing of the SSD firmware stack in the context of the host application and driver stack greatly increases the embedded firmware quality.

Further important use-cases are scripting, regression, and continuous integration to automate the testing of the SSD firmware, as well as coverage-driven fault injection to further increase the SW quality beyond the expected application context.

Accurate Virtual Prototype of SSD SoC

The Loosely Timed (LT) platform of an SSD controller is optimized for high simulation speed and bit-accurate behavior but does not model the detailed timing. To enable performance analysis and optimization of the firmware and the SoC architecture, the fast functional models of the processor, the interconnect, and the DRAM memory controller are replaced with their accurate counterparts. The remaining models are not in the critical path of the performance analysis and stay at the LT level. The timing annotation of the LT flash controller model is sufficiently accurate to model the delay of the Flash operations. To simplify the directed performance testing, the PCIe and NVMe host interfaces are removed.

Instead, the host commands are poked into the CPU memory space using simulation scripts to directly trigger the respective FTL operation.

In this example, the instruction accurate ISS of the Arm CPU is replaced by the corresponding Arm Cycle Model (Arm Cycle Models). Then the flat untimed LT memory is exchanged with an accurate model of the Synopsys DesignWare uMCLT2 DDR memory controller. The last step is to replace the simple LT bus model with a cycle accurate model of Arm CoreLink NIC400. Both the uMCTL2 and NIC400 TLM performance models are available in the model library of Synopsys Platform Architect, a Virtual Prototyping environment for architecture use-cases (Synopsys Platform Architect). Based on this incremental refinement, the Loosely Timed SW-VP is converted into a Virtual Prototype for architecture analysis. The resulting architecture model of the SSD controller is shown on the left side of Fig. 16.

The main value of running a timing accurate Virtual Prototype is provided by the analysis results generated by the simulation. As an example, the charts in the middle of Fig. 16 show from top to bottom the event statistics of the CPU Performance Monitor Unit (PMU), the firmware function trace, the bus throughput, and the data channel utilization of the DDR memory. Especially the PMU analysis provides great insight into how efficiently the CPU subsystem executes the specific firmware, e.g., the performance of the branch predictor, the different reasons for CPU pipeline stalls, or the hits and misses in the level one instruction and data caches.

This level of detail is essential for root cause analysis, to identify and fix the reasons for performance issues, either in the firmware or the HW architecture. However, looking at the results of one simulation at a time is not efficient for analyzing design trade-offs. The latter is more effectively done using sensitivity analysis with pivot charts as shown on the right side of Fig. 16. Here the aggregated KPIs from a parameter sweep can be compared, highlighting the impact of design and configuration parameters on various metrics.

The specific example shows the execution time for the boot sequence (blue bars) and for processing 10 host commands (orange bars), depending on a variety of settings for the CPU clock frequency, the DDR memory speed, and the depth of the bus transaction pipeline. The pivot chart shows a saturation of the performance at the lower as well as the higher end of the spectrum, and a clear performance increase in the middle.

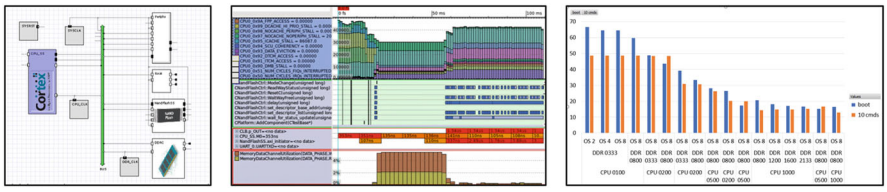


Fig. 16 Virtual Prototype for performance analysis of SSD controller (left), detailed analysis results (middle), and sensitivity analysis (right)

Similar analysis would be possible with the RTL model of the SSD controller, but only at a much later point in time of the development process. The cycle accurate TLM model provides much better visibility for HW and SW performance analysis and faster turn-around time for parametric what-if analysis.

SSD Case Study Summary

This section summarizes the benefits of Virtual Prototypes in the context of this SSD controller case study.

The SW-VP is first used for the bring-up of the SSD firmware stack, in particular the porting of the device drivers of the NAND flash controller and the PCIe host interface. The PCIe Virtual I/O concept further expands the scope for firmware development and testing on Virtual Prototypes. It enables the validation in the context of real-world applications for PCIe end point devices, like SSD controllers, Smart Network Interface Cards (SmartNIC), or AI accelerator cards. This expands the range of available test scenarios, enabling host side and device side software to be built, run, and tested in the target environment. Similar benefits apply for the virtual and real-world connectivity of other interface IP, like USB or Ethernet.

Refining the Loosely Timed SW-VP into timing accurate VP enables the joint hardware and software performance analysis and optimization. Especially the analysis of performance counters in cycle accurate processor models provides insight into cache performance and micro-architecture effects like branch prediction and pipeline stalls. The accurate traffic can be used for interconnect and memory performance optimization.

The best return of investment comes from taking advantage of both software and architecture use-cases. Part of the TLM models can be re-used between the loosely timed and timing accurate Virtual Prototype to reduce the modeling effort. Having the firmware running on the SW-VP as a known good starting point greatly increases the productivity for the creation and bring-up of the more accurate and slower performance analysis platform.

Conclusion and Outlook

This chapter reviews the state of the art in Virtual Prototyping, with a focus on use-cases around early architecture analysis and software development, including an introduction to the underlying SystemC based TLM modeling paradigm.

The use of Virtual Prototypes for architecture analysis during the specification phase is growing continuously, as it becomes increasingly difficult to predict the non-functional properties like power and performance of heterogeneous many-processor SoCs. Here, Virtual Prototypes help to reduce the risk of under-designing and over-designing, especially as chip design projects increasingly require deep collaboration of different teams and even across multiple companies.

On top of the growing complexity of individual chips, the architecture analysis needs cope with the specification of multi-chip systems, either in the form of multiple chiplets in one package, or in the form of distributed compute functions such as data-center AI training and AI inference (Novet 2021). This drives the need for higher levels of abstractions and consideration of chip-to-chip protocols like die-to-die communication, PCIe, Ethernet, or photonic interconnects.

The deployment of machine learning is a general trend in EDA in order to automate design steps and to improve quality of results (Huang et al. 2021). As the design space increases, techniques like reinforcement learning can help with the architecture exploration of complex SoCs, especially when the target application domain requires near optimal results (Krishnan et al. 2021).

Today the broadest adoption of Virtual Prototypes is the area of early software development, where Loosely Timed SW-VPs enable tasks like developing, integrating, and testing of hardware dependent software before silicon or even RTL become available. This greatly mitigates the schedule risk for projects with new software and IP components. The use of coverage-based fault injection ensures the quality and robustness of the embedded software for safety critical applications. The Loosely Timed SW-VP can also be used as a tool for early go-to-market and customer enablement by sharing the VP-based software development environment with the device maker.

A current trend in embedded SW development is to leverage general software engineering practices like agile development and DevOps, aiming to enable the continuous delivery of new features with high quality. For this purpose, HW based testing is complemented with SW-VPs for building an automated embedded DevOps environment. This extends the live cycle of a SW-VP beyond the early pre-RTL/pre-silicon phase towards post-silicon testing of embedded SW updated as part of a continuous integration and deployment chain.

A perpetual trend is the continuous quest for raising the level of abstraction to cope with the ever increasing complexity of developing embedded systems and systems of systems. This drives the need for more high-level abstraction techniques like device virtualization (The Software Freedom Conservancy), OS-level simulation (Silver Virtual ECU), and surrogate models (Barve et al. 2021). It will still be important to mix and match different levels of abstraction, since software development tasks like debugging and testing require a certain level of fidelity, at least for the respective point of observation.

Recently the term *Digital Twin* (Fuller et al. 2020) has become fashionable, especially in the context of automotive and defense/aerospace applications. A digital twin goes beyond the modeling of an embedded system by enabling continuous exchange of data between the physical system and the virtual counterpart. Virtual Prototypes have the potential to play a part in such a Digital Twin setup, provided they can deliver the necessary simulation speed.

Trademarks

Synopsys, Virtualizer, Platform Architect, and DesignWare are registered trademarks of Synopsys, Inc.

Arm, Cortex, and AMBA are registered trademarks of Arm Limited. “Arm” is used to represent Arm Holdings plc.; its operating company Arm Limited; and its regional subsidiaries.

All other brands or product names are the property of their respective holders.

Disclaimer

The opinions and observations expressed in this publication are my own. They do not purport to reflect the opinions, views, or intentions of my employer Synopsys, Inc.

References

Android Studio. https://developer.android.com/studio
Arm Cycle Model Studio. https://www.arm.com/products/development-tools/simulation/cycle-model-studio
Arm Cycle Models. https://developer.arm.com/tools-and-software/simulation-models/cycle-models
Arm Development Studio. https://www.arm.com/products/development-tools
Arm Fast Models. https://developer.arm.com/tools-and-software/simulation-models/fast-models
Arm Fixed Virtual Platforms. https://developer.arm.com/tools-and-software/simulation-models/fixed-virtual-platforms
ASIP Designer. https://www.synopsys.com/designware-ip/processor-solutions/asips-tools.html
CEVA. https://www.ceva-dsp.com
DesignWare ARC nSIM. https://www.synopsys.com/dw/ipdir.php
Functional Mock-up Interface (FMI). https://fmi-standard.org
Intel CoFluent. https://www.intel.com/content/www/us/en/cofluent/overview.html
Intel Integrated Simulation Infrastructure with Modeling (ISIM). https://software.intel.com/content/www/us/en/develop/tools/integrated-simulation-infrastructure.html
Jenkins Automation Server for Continuous IntegrationContinuous Delivery. https://www.jenkins.io/
Lauterbach Development Tools. https://www.lauterbach.com
Mirabilis VisualSim. https://www.mirabilisdesign.com
Oracle Virtual Box. https://www.virtualbox.org
Silver Virtual ECU. https://www.synopsys.com/verification/virtual-prototyping/virtual-ecu/silver.html
Simulink. https://www.mathworks.com/products/simulink.html
Synopsys DesignWare TLM library. https://www.synopsys.com/verification/virtual-prototyping/virtual-prototyping-models/designware-tlm-library.html
Synopsys Platform Architect. https://www.synopsys.com/verification/virtual-prototyping/platform-architect.html
Synopsys Virtualizer. https://www.synopsys.com/verification/virtual-prototyping/virtualizer.html
Tensilica Customizable Processors. https://ip.cadence.com/ipportfolio/tensilica-ip
The Modelica Association. https://modelica.org
The OpenSSD Project. http://www.openssd.io
The Software Freedom Conservancy. QEMU the fast processor emulator. https://www.qemu.org
The SystemC Modeling Library (SCML). http://www.synopsys.com/cgi-bin/slcw/kits/reg.cgi
Tool CommLanguage (TCL). http://www.tcl.tk
Verilator. https://www.veripool.org/verilator
IEEE Standard for Standard SystemC Language Reference Manual (2012) IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005), pp 1–638

AMBA AXI and ACE Protocol Specification (2013) http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022e/index.html
IEEE Standard for Design and Verification of Low-Power Integrated Circuits (2015) http://standards.ieee.org/getieee/1801/download/1801-2015.pdf
Accelerate devops with continuous integration and simulation, a how-to guide for embedded development (2019) https://resources.windriver.com/devops/accelerate-devops-with-continuous-integration-and-simulation
IEEE Standard for Universal Verification Methodology Language Reference Manual (2020) IEEE Std 1800.2-2020 (Revision of IEEE Std 1800.2-2017), pp 1–458
Aldis J (2006) Use of systemC modelling in creation and use of an SOC platform: experiences and lessons learnt from OMAP-2. In: Burton M, Morawiec A (eds) Platform based design at the electronic system level. Springer, New York, pp 31–47
Arm Ltd (2020) AMBA-PV Extensions to TLM Developer Guide, Version 2.0. https://developer.arm.com/documentation/100962/latest
Balarin F, Chiodo M, Giusto P, Hsieh H, Jurecska A, Lavagno L, Passerone C, Sangiovanni-Vincentelli A, Sentovich E, Suzuki K, Tabbara B (1997) Hardware-software co-design of embedded systems: the POLIS approach. Springer, Boston
Barve Y, Karve P, Gokhale A, Mahadevan S (2021) Research challenges in the design and composition of surrogate models for robust CPS: position paper. Association for Computing Machinery, pp 26–29
Becker D, Singh RK, Tell SG (1992) An engineering environment for hardware/software co-simulation. In: Proceedings of the 29th ACM/IEEE design automation conference. IEEE Computer Society Press, pp 129–134
Bhote P, Kallerdahl A, Khan O, Hardt W (2019) Latest trends in continuous integration for highly autonomous driving. In: IBS international symposium on computer science, computer engineering and educational technology 2019, p 49
Binkert N, Beckmann B, Black G, Reinhardt SK, Saidi A, Basu A, Hestness J, Hower DR, Krishna T, Sardashti S et al (2011) The gem5 simulator. ACM SIGARCH Comput Archit News 39(2):1–7
Buck J, Ha S, Lee EA, Messerschmitt D (1994) Ptolemy: a framework for simulating and prototyping heterogeneous systems. Int J Comput Simul 4:155–182
DeSchutter T (ed) (2014) Better software. Faster! best practices in virtual prototyping. Synopsys . ISBN 978-1617300134
Edwards S, Lavagno L, Lee E, Sangiovanni-Vincentelli A (1997) Design of embedded systems: formal models, validation, and synthesis. Proc IEEE 85(3):366–390
Engblom J (2018) Temporal decoupling – are fast and correct mutually exclusive? In: Design and verification conference in Europe (DVCon). https://dvcon-europe.org/content/event-details?id=260-5
Feldner I, Heer C, Kogel T, Mauderer A, Schleifer R, Thanner M, Thudt R (2019) The automotive virtual prototyping platform. Virtual Platform Working Group of the Arbeitskreis Automotive. https://www.edacentrum.de/en/whitepaper-automotive-virtual-prototyping-platform
Flynn D, Aitken R, Gibbons A, Shi K (2007) Low power methodology manual: for system-on-chip design. Springer, New York
Friedman J (2006) MATLAB/simulink for automotive systems design. In: Proceedings of the design automation test in Europe conference, vol 1, pp 1–2
Fuller A, Fan Z, Day C, Barlow C (2020) Digital twin: enabling technologies, challenges and open research. IEEE Access 8:108952–108971
Gajski DD, Vahid F, Narayan S, Gong J (1998) System-level exploration with specSyn. In: Proceedings of the 35th annual design automation conference, DAC'98. Association for Computing Machinery, New York, pp 812–817
Greaves DJ (2021) Modern system-on-chip design on arm. Arm Education Media. ISBN 978-1-911531-37-1
Groetker T, Liao S, Martin G, Swan S (2002) System design with systemC. Springer, Heidelberg

Guerra L, Fitzner J, Talukdar D, Schlager C, Tabbara B, Zivojnovic V (1999) Cycle and phase accurate DSP modeling and integration for HW/SW co-verification. In: Proceedings 1999 design automation conference (Cat. No. 99CH36361), pp 964–969

Gupta RK, De Micheli G (1993) Hardware-software cosynthesis for digital systems. IEEE Des Test 10(3):29–41

Hellestrand G (1999) The revolution in systems engineering. IEEE Spectr 36(9):43–51

Hoffmann A, Kogel T, Nohl A, Braun G, Schliebusch O, Wahlen O, Wieferink A, Meyr H (2001) A novel methodology for the design of application-specific instruction-set processors (ASIPS) using a machine description language. IEEE Trans Comput-Aided Des Integr Circuits Syst 20(11):1338–1354

Huang G, Hu J, He Y, Liu J, Ma M, Shen Z, Wu J, Xu Y, Zhang H, Zhong K, Ning X, Ma Y, Yang H, Yu B, Yang H, Wang Y (2021) Machine learning for electronic design automation: a survey

Jünger L, Zurstraßen N, Kogel T, Keding H, Leupers R (2020) Amaix: a generic analytical model for deep learning accelerators. In: Orailoglu A, Jung M, Reichenbach M (eds) Embedded computer systems: architectures, modeling, and simulation. Springer International Publishing, Cham, pp 36–51

Kang K, Park S, Bae B, Choi J, Lee S, Lee B, Lee JB (2019) Seamless SoC verification using virtual platforms: an industrial case study. In: Design, automation test in Europe conference exhibition (DATE), pp 1204–1205

Kempf T, Dörper M, Leupers R, Ascheid G, Meyr H, Kogel T, Vanthournout B (2005) A modular simulation framework for spatial and temporal task mapping onto multi-processor soc platforms. In: Proceedings of the conference on design, automation & test in Europe (DATE), Munich

Keutzer K, Newton A, Rabaey J, Sangiovanni-Vincentelli A (2000) System-level design: orthogonalization of concerns and platform-based design. IEEE Trans Comput-Aided Des Integr Circuits Syst 19(12):1523–1543

Kienhuis B, Deprettere E, Vissers K, Van Der Wolf P (1997) An approach for quantitative analysis of application-specific dataflow architectures. In: Proceedings IEEE international conference on application-specific systems, architectures and processors, pp 338–349

Kogel T (2006) Peripheral modeling for platform driven ESL design. In: Burton M, Morawiec A (eds) Platform based design at the electronic system level. Springer, New York, pp 71–85

Kogel T (2017) Synopsys virtual prototyping for software development and early architecture analysis. In: Ha S, Teich J (eds) Handbook of hardware/software codesign. Springer, Dordrecht, pp 1127–1159

Krishnan S, Wan Z, Bharadwaj K, Whatmough P, Faust A, Neuman S, Wei GY, Brooks D, Reddi VJ (2021) Autopilot: automating SoC design space exploration for swap constrained autonomous UAVs

Lapedus M (2018) Big trouble at 3 nm. https://semiengineering.com/big-trouble-at-3nm

Lecler JJ, Baillieu G (2011) Application driven network-on-chip architecture exploration & refinement for a complex SoC. Des Autom Embed Syst 15(2):133–158

Liao S, Tjiang S, Gupta R (1997) An efficient implementation of reactivity for modeling hardware in the scenic design environment. In: Proceedings of the 34th design automation conference, pp 70–75

Liebel G, Marko N, Tichy M, Leitner A, Hansson J (2018) Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice. Softw Syst Model 17(1):91–113

Martin G (1998) Design methodologies for system level IP. In: Proceedings of the conference on design, automation and test in Europe. IEEE Computer Society, pp 286–289

Mäenpää M (2020) Virtualized CPU usage in SoC verification. Master's thesis, University of Oulu, Faculty of Information Technology and Electrical Engineering. http://urn.fi/URN:NBN:fi:oulu-202008282897

Micheloni R, Marelli A, Eshghi K (2018) Inside solid state drives (SSDs). Springer series in advanced microelectronics. Springer, Singapore. https://books.google.de/books?id=UtNjDwAAQBAJ

Mueller W, Becker M, Elfeky A, DiPasquale A (2012) Virtual prototyping of cyber-physical systems. In: 17th Asia and South Pacific design automation conference, pp 219–226

Nohl A, Braun G, Schliebusch O, Leupers R, Meyr H, Hoffmann A (2002) A universal technique for fast and flexible instruction-set architecture simulation. In: Design automation conference

Novet J (2021) Tesla unveils chip to train A.I. models inside its data centers. https://www.cnbc.com/2021/08/19/tesla-unveils-dojo-d1-chip-at-ai-day.html

Oetjens JH, Bannow N, Becker M, Bringmann O, Burger A, Chaari M, Chakraborty S, Drechsler R, Ecker W, Grüttner K, Kruse T, Kuznik C, Le HM, Mauderer A, Müller W, Müller-Gritschneder D, Poppen F, Post H, Reiter S, Rosenstiel W, Roth S, Schlichtmann U, von Schwerin A, Tabacaru BA, Viehl A (2014) Safety evaluation of automotive electronics using virtual prototypes: state of the art and research challenges. In: 2014 51st ACM/EDAC/IEEE design automation conference (DAC), pp 1–6

Reyes V (2013) Virtual hardware in-the-loop: earlier testing for automotive applications. https://www.eenewseurope.com/Learning-center/virtual-hardware-“-loop”-earlier-testing-automotive-applications

Rowson J (1997) Virtual prototyping. In: Proceedings of CICC 97 – custom integrated circuits conference, pp 89–94

Rowson J, Sangiovanni-Vincentelli A (1997) Interface-based design. In: Proceedings of the 34th design automation conference, pp 178–183

Schürmans S, Zhang D, Auras D, Leupers R, Ascheid G, Chen X, Wang L (2013) Creation of ESL power models for communication architectures using automatic calibration. In: 2013 50th ACM/EDAC/IEEE design automation conference (DAC), pp 1–6

Song YH, Jung S, Lee SW, Kim JS (2014) Cosmos openssd: a pcie-based open source ssd platform. In: Proceedings of flash memory summit

Van Rompaey K, Verkest D, Bolsens I, De Man H (1996) Coware-a design environment for heterogeneous hardware/software systems. In: Proceedings EURO-DAC’96. European design automation conference with EURO-VHDL’96 and exhibition, pp 252–257

Wolf W (2003) A decade of hardware/software codesign. Computer 36(4):38–43

FPGA-Specific Compilers

28

Nitish Srivastava, Gai Liu, Yi-Hsiang Lai, and Zhiru Zhang

Contents

Introduction . . . 990
Existing HLS Compilers and Programming Models . . . 992
C-Based HLS Tools . . . 993
Dataflow Compilers . . . 994
Domain-Specific Languages (DSLs) . . . 995
Emerging Accelerator Design Languages . . . 996
Key Compiler and Synthesis Optimizations . . . 997
Pipelining Techniques . . . 998
Parallelization Techniques . . . 1004
Memory Customization Techniques . . . 1007
Data Type Customization Techniques . . . 1011
Case Study: Binarized Convolutional Neural Networks . . . 1013
Algorithm Overview . . . 1013
Pipelining and Unrolling . . . 1014
Line Buffers and Window Buffers . . . 1016
Data Vectorization . . . 1016

Equal contribution from the first two authors; part of their work was done at Cornell.

N. Srivastava
Google LLC, Mountain View, CA, USA

G. Liu
Xilinx, Inc., San Jose, CA, USA
e-mail: gai.liu@xilinx.com

Y.-H. Lai · Z. Zhang (✉)
Cornell University, Ithaca, NY, USA
e-mail: yl2666@cornell.edu; zhiruz@cornell.edu

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_25

Building the BNN Accelerator Using HeteroCL 1017
Evaluation 1017
Concluding Remarks 1019
References 1020

Abstract

Recent years have seen a rising popularity of FPGA-based compute acceleration as well as significant advances in the compilation tools that aim to make FPGAs accessible to software programmers. Modern FPGA-specific compilers, especially in the form of high-level synthesis (HLS) tools, have been increasingly used to automatically generate optimized accelerators from software programs. In this chapter the authors begin by surveying contemporary HLS compilers, followed by an anatomy of factors affecting the throughput of a custom accelerator, i.e., parallelism, utilization, and frequency. The authors then discuss four major categories of optimization techniques that are commonly used in HLS to generate high-performance accelerators, including pipelining, parallelization, memory customization, and data type customization. For each category, the authors survey a subset of the optimization techniques found in the recent literature and commercial tools. In the end, the authors use a binarized neural network as a case study to demonstrate the usage and benefits of these techniques.

Keywords

Compiler · Domain-specific language (DSL) · Field-programmable gate array (FPGA) · Hardware acceleration · High-level synthesis (HLS) · Parallelization · Pipelining

Introduction

Over the past two decades, field-programmable gate arrays (FPGAs) have evolved from small chips with a few thousand logic blocks to billion-transistor systems-on-chip that offer an attractive option for flexible and efficient accelerated computing. In contrast to general-purpose processors (CPUs) and graphics processing units (GPUs), FPGAs can be reconfigured to implement a highly specialized accelerator architecture that is specifically optimized based on the key characteristics of a target application. More specifically, the compute pipeline, memory hierarchy, and the numerical representation of an FPGA accelerator are all customizable (Choi et al. 2016). For many applications, having such architectural flexibility can overcome the limitation of a slower clock on an FPGA device, leading to both higher performance and improved energy efficiency.

However, software programmability is a major hurdle for the wide deployment of FPGA-based acceleration (Bacon et al. 2013). Traditionally, FPGAs are programmed using register-transfer level (RTL) languages where the designer is

directly responsible for specifying the low-level cycle-accurate behavior of the underlying accelerator architecture. This is a very time-consuming and labor-intensive process. Worse, the manual RTL design cannot be easily ported to different FPGA devices. Since the mid 2000s, there has been extensive research and development towards making FPGAs accessible to software-inclined developers, besides hardware specialists. Many programming models and automated synthesis tools have been proposed to tackle this grand challenge. In particular, modern high-level synthesis (HLS) tools have emerged as a promising alternative to the RTL design methodology to enable productive design and implementation of the FPGA-based accelerators (Cong et al. 2011). HLS raises the level of input abstraction from hardware to software by automatically transforming a software description in a high-level programming language such as C/C++/OpenCL to a low-level RTL design in an HDL like Verilog or VHDL. It allows designers to more easily explore a rich set of accelerator architectures through automated synthesis and user-specified directives that direct the compiler to implement a variety of hardware customizations.

Unlike CPUs and GPUs, most FPGA accelerators have application-specific architectures and do not use a pre-defined instruction set. Hence compiling software programs onto the FPGA fabrics poses many unique challenges. First, an FPGA compiler (e.g., HLS tool) must infer a specialized and high-performance accelerator architecture (e.g., deeply pipelined and massively parallel) either automatically or from hints given by user directives. Second, modern FPGA devices feature a heterogeneous mix of compute and memory resources such as look-up tables (LUTs), hard DSP blocks, and SRAM blocks with various capacities; the on-chip communication networks are also fully customizable (Gaide et al. 2019). Therefore, the compiler needs to effectively exploit both application-level characteristics and the hardware heterogeneity to maximize the performance of the design by fully utilizing the rich variety of FPGA resources. Third, the FPGA devices tend to operate at a lower but more flexible range of clock frequencies (typically 150–450 MHz for modern device families). This exposes the compiler to a much wider trade-off space that involves frequency, throughput, and resource utilization. For example, the compiler can either choose to schedule multiple operations within one cycle in a combinational way (also known as operation chaining) or create a multi-level pipelined datapath. Operation chaining may lower clock frequency, but potentially increases resource utilization which in turn increases hardware parallelism. Last but not least, it is vital for an FPGA compiler to support parameterized numeric types and automatic bitwidth analysis techniques. Exploiting custom data representations can lead to substantially improved frequency and resource usage.

▶ Chapter 24, "Accelerator Design with High-Level Synthesis" discussed a general introduction to HLS principles for both ASICs and FPGAs and presented the related problems more from an HLS tool developer's perspective. In this chapter the authors provide a discussion on FPGA-specific compilers more from a user's perspective, where the focus is on different programming and optimization techniques that are essential for developing a high-performance FPGA accelerator. The authors survey the recent HLS tools as well as the essential compilation and

synthesis techniques that enable software-defined FPGA acceleration. The authors particularly focus on the techniques that are unique to custom accelerator designs, instead of the well-known code optimizations that are established for CPU or GPU targets. The rest of the chapter is organized as follows: section "Existing HLS Compilers and Programming Models" surveys a representative set of existing HLS compilers; section "Key Compiler and Synthesis Optimizations" outlines the key compiler and synthesis optimizations and focuses on three main pillars of high-performance FPGA design; sections "Pipelining Techniques" and "Parallelization Techniques" describe commonly used pipelining and parallelization techniques at different levels of design granularity; section "Memory Customization Techniques" introduces popular memory optimization approaches and how the language features supported by FPGA-targeted compilers can be used to implement them; section "Data Type Customization Techniques" discusses FPGA-specific data type optimizations; section "Case Study: Binarized Convolutional Neural Networks" provides a case study of implementing a binarized convolutional neural network followed by concluding remarks in section "Concluding Remarks".

Existing HLS Compilers and Programming Models

In this section, the authors describe and categorize the existing HLS tools and compilers targeting FPGAs. There is a broad spectrum of programming models and compilers for FPGAs, such as C-based HLS, dataflow compilers, domain-specific languages, and emerging accelerator design languages. Due to the space limitation, the authors can only survey a (small) subset of the representative and more recent efforts.

While modern HLS tools may differ significantly in their input specifications, they usually follow a similar design flow, which the authors sketch in Fig. 1. Starting from a software program, the designer manually partitions it into a software part running on the host CPU, and a set of compute-intensive kernels that are offloaded to the FPGA. Some of the hardware/software partitioning tasks can also be done automatically (King and Dave 2012) . The designer further provides the tool with pragmas and directives to instruct the tool to generate the desired datapath and memory components. As described in the ▸ Chap. 24, "Accelerator Design with High-Level Synthesis", the HLS tool first compiles the source level program into an intermediate representation (IR), typically using an off-the-shelf software compiler front-end. The IR is then iteratively transformed and optimized using native and FPGA-specific passes. After that, compute kernels and tasks are extracted by the tool to expose the data-level and task-level parallelism. Each of the kernels and tasks is optimized through scheduling, pipelining, and other optimizations to extract loop-level and operator-level parallelism. In addition, customized memory hierarchies are generated to provide sufficient memory bandwidth to the compute kernels. Finally, an RTL design is generated and synthesized into an FPGA bitstream.

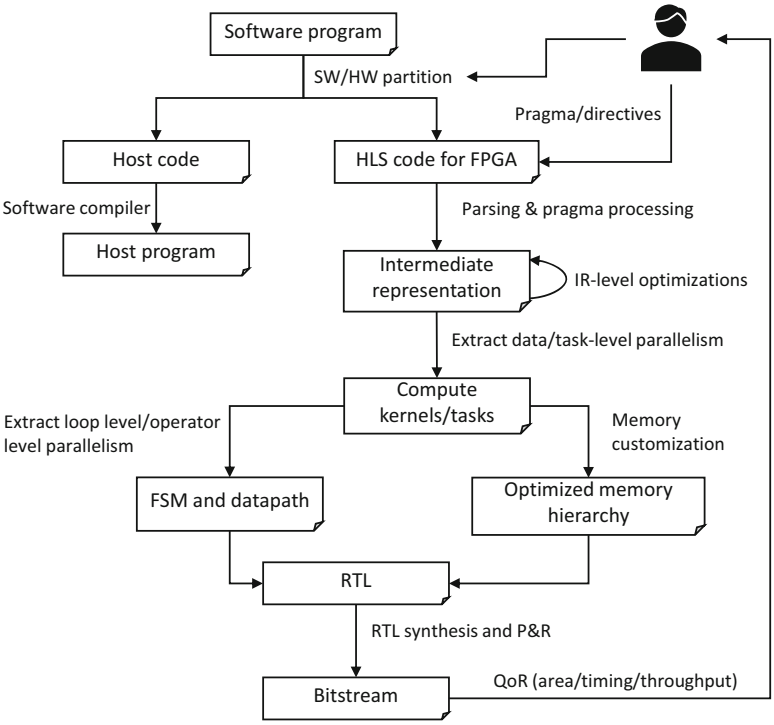


Fig. 1 A typical design flow with high-level synthesis

C-Based HLS Tools

The contemporary HLS compilers are commonly based on C/C++ and its extensions (e.g., OpenCL, SystemC). These tools accept sequential C-like code as input with optional user directives, and generate optimized hardware implementations on the FPGA by exploiting various parallelization and customization opportunities either automatically or guided by user directives. The representative C-based commercial HLS tools include Xilinx Vivado/Vitis HLS, Xilinx Vitis Unified Software Platform, Intel FPGA SDK for OpenCL, and Intel HLS Compiler. The HLS compilers based on C/C++ have the advantage of allowing programmers to express algorithms in an imperative way using the familiar C semantics, while leaving the work of extracting parallelism and memory specialization to the compiler. Tools such as the Xilinx OpenCL Compilerand Intel FPGA SDK for OpenCLcan generate optimized hardware implementations exploiting the parallelism explicitly expressed in the OpenCL language. Such tools are especially suitable for applications with a high degree of regular data-level parallelism. Mentor Catapultand Cadence StratusHLS tools mainly focus on ASIC designs, although they can also target a number of popular FPGA devices. LegUp (Canis et al. 2011), Bambu (Pilato and Ferrandi

2013), ROCCC (Najjar et al. 2016), GAUT (Coussy et al. 2008), and Kiwi Compiler (KiwiC) (Singh and Greaves 2008) are among the well-known open-source HLS tools developed by academic groups.

One of the key challenges of C-based HLS tools is for a user to write "hardware-friendly" code in a way that efficient parallel/pipelined architecture can be inferred from a sequential program. A newer generation of tools try to automate this process to ease the burden of the programmers. For example, the Merlin Compiler (2020) from Falcon Computing (recently acquired by Xilinx) takes OpenMP-style C programming model and automatically generates the HLS C/OpenCL code with optimized off-chip data movement, on-chip data reuse and memory partitioning, and parallel-pipelined loops. Silexica (also acquired by Xilinx) provides the SLX-FPGA Tool Suiteto help convert non-synthesizable C/C++ code to synthesizable HLS C code for Xilinx Zynq SoC and MPSoC devices. Delft Workbench (DWB) (Nane et al. 2014) is a toolchain that uses Quipu (Meeuws et al. 2011) and Quad (Ostadzadeh et al. 2010) to predict the hardware usage and memory accesses of a high-level application written in C and maps the compute-intensive functions onto FPGAs.

Dataflow Compilers

Dataflow programming was initially proposed in the domain of parallel computing on CPUs (Dennis 1974) and has later emerged as a viable programming model for FPGAs (Putnam et al. 2008). In this model, an application is represented as a dataflow graph where the vertices represent computation processes and edges represent communication channels between the processes as discussed in the ▸ Chap. 24, "Accelerator Design with High-Level Synthesis" CAL Actor (Eker and Janneck 2003) is one of the pioneering works in data flow languages. CAL allows the programmer to express the functionality of dataflow actors, a computation on sequences of tokens (atomic pieces of data) that produces other sequences of tokens as a result, in a textual form. The dataflow-based model of computation is particularly suitable for describing spatial architectures on FPGAs. For FPGAs, the vertices in the dataflow graph are mapped to the compute resources, and the edges are typically implemented with FIFOs or shift registers (using flip-flops, LUT RAMs, and/or BRAMs). Thus there are specialized HLS tools that build on the dataflow abstraction, where the designer has the explicit control of the processes and interconnections in the dataflow network. CHiMPS (Putnam et al. 2008) is a C-based dataflow HLS compiler that translates a C program into CHiMPS target language (CTL) instruction blocks and instantiates each node in a program's dataflow graph as a physical node in the resulting FPGA accelerator.

MaxCompiler (Lindtjorn et al. 2011) is a compilation tool from Maxeler Technologies for their proprietary FPGA-based dataflow engines (DFEs). The compute-intensive portions of the application are written using a Java meta-language named MaxJ . OXiGen (Peverelli et al. 2018) is a very recent hardware compilation tool that

takes a compute-intensive C function and translates it to a dataflow representation for the MaxCompiler. Open DataFlow (OpenDF) (Janneck et al. 2011) is an open-source initiative of Xilinx, where an algorithm is defined as a dataflow graph and the actors are written using CAL Actor Language. StreamBlocks (Bezati et al. 2020), based on the Tycho compiler infrastructure, transforms each actor in a dataflow program to an abstract machine model which provides a unified model for executing actors in both hardware and software. CAPH (Sérot et al. 2013) is a dataflow language for describing, simulating, and implementing stream-processing applications on FPGAs.

Domain-Specific Languages (DSLs)

DSLs provide more specialized language constructs and the associated compilation flow that target a specific domain. This raises the level of abstraction for the programmers and potentially simplifies the work of compilers in identifying and exploiting opportunities for advanced domain-specific optimizations. PolyMage (Mullapudi et al. 2015) includes a Python-embedded image processing DSL and a polyhedral compilation framework composed of an optimizer and an autotuner. It can automatically generate high-performance image processing pipelines executed on the reconfigurable hardware. Halide (Ragan-Kelley et al. 2013) is a DSL specifically designed for high-performance image processing for CPUs/GPUs. Halide-HLS (Pu et al. 2017), HeteroHalide (Li et al. 2020), and GENESIS (Ishikawa et al. 2019) build on Halide to generate optimized image processing pipelines for FPGAs. In addition to major works exploiting the Halide compiler within their toolchain, there are a number of projects such as Liao et al. (2019) and Carlson and Van Wyk (2019) that follow the same workflow but with either extended versions of the Halide compiler or Halide-inspired DSL compilers to support their own domain-specific structures. Darkroom (Hegarty et al. 2014) and Rigel (Hegarty et al. 2016) can capture image processing algorithms as DAGs of basic image processing operations and generate efficient hardware accelerators for FPGA. Heterogeneous image processing acceleration (Hipacc) (Reiche et al. 2017) is another DSL that is able to produce low-level code for image processing kernels on FPGAs. The Rathlin image processing language (RIPL) (Stewart et al. 2018) is a DSL for developing memory-efficient image processing applications.

OptiML (Sujeeth et al. 2011), a Scala-embedded machine learning DSL implemented using the Delite compiler framework (Lee et al. 2011), is an automated design tool for realizing FPGA accelerators from high-level programs. TensorFlow (Abadi et al. 2016), MXNet (Chen et al. 2015), Caffe (Jia et al. 2014), and PyTorch (Paszke et al. 2019) are some of the common DSLs designed specifically for deep learning. Caffeine (Zhang et al. 2018a), based on Caffe, is a hardware/software co-design framework to efficiently accelerate the entire CNN on FPGAs. Spatial Multiverse (Hadjis and Olukotun 2019) converts a TensorFlow trained model to Spatial (Koeplinger et al. 2018) hardware IR and generates

bitstreams for FPGA. TVM (Chen et al. 2018) is a Halide-inspired compilation framework for deep learning compilation. It exposes graph-level and operator-level optimizations and maps deep learning workloads across diverse hardware backends, including FPGAs (Moreau et al. 2018). DNNBuilder (Zhang et al. 2018b) is an automated tool flow that can transform DNN designs from popular deep learning frameworks such as Caffe and TensorFlow to highly optimized board-level FPGA implementations.

T2S-Tensor (Srivastava et al. 2019) is a Halide-based DSL that allows the programmers to specify a dense tensor computation in functional notation along with decoupled spatial optimization directives. It generates high-performance systolic arrays for FPGAs and CGRAs. SuSy (Lai et al. 2020) allows programmers to specify the systolic algorithms in the form of uniform recurrence equations (UREs) and compiles them to FPGAs. T2S-Tensor and SuSy belong to the family of T2S DSLs (Rong 2017, 2018a,b) that have been proposed to productively generate high-performance systolic array accelerators.

Emerging Accelerator Design Languages

While DSLs offer many advantages in productivity and compilation for individual application domains, more general-purpose language models are still needed to (1) bridge the gaps between popular domains, (2) provide programmers with greater control on important optimizations, (3) and serve as a compilation target for multiple high-level DSLs. Hence there is an increasingly popular trend to raise the level of abstraction for HLS designs, while still being able to target various application domains. Emerging accelerator design languages are specialized to abstract away the implementation-level details of a C-based HLS design, while allowing the designer to focus on higher-level design and optimization decisions. Spatial (Koeplinger et al. 2018) is a Scala-based language and compiler to define hardware accelerators. It is built using the Delite hardware definition language (DHDL) (Koeplinger et al. 2016). Hot & Spicy (Skalicky et al. 2018) is an open-source framework and toolchain for exploiting FPGA accelerators in applications developed completely in Python. HeteroCL (Lai et al. 2019) is composed of a Python-based DSL and an automated compilation flow that maps the input algorithm into special-purpose accelerators through HLS. Similar to Halide (Ragan-Kelley et al. 2013) and TVM (Chen et al. 2018), HeteroCL separates an algorithm specification from a temporal compute schedule. It further decouples the algorithm from memory architectures and data quantization schemes, which are both essential for efficient hardware customization. With respect to memory customization, HeteroCL provides primitives to create custom memory hierarchy through banking, reuse buffers, and data streaming. Dahlia (Nigam et al. 2020) is a new HLS language that uses a type system to restrict the design space to HLS programs that can be predictably compiled to hardware accelerators.

Key Compiler and Synthesis Optimizations

While the existing HLS tools introduced in section "Existing HLS Compilers and Programming Models" may differ significantly in their input specifications and targeted application domains, they often employ a common set of optimizations to achieve the design's performance target. This section summarizes such key optimizations that are ubiquitous in the existing tools and analyzes how each of them affects the overall design throughput.

There are three major factors that impact the throughput of an HLS-based design, which the authors summarize into the following formula:

$$Throughput \propto Parallelism \times Utilization \times Frequency \quad (1)$$

Parallelism refers to the number of parallel compute units instantiated on the FPGA hardware that processes parallelizable tasks simultaneously. Assuming these compute units are fully utilized during run time, the system throughput is proportional to its parallelism. **Utilization** is the percentage of the time when each compute unit is executing meaningful tasks (as opposed to idling or being blocked). A higher utilization improves the design throughput without increasing the resource usage. **Frequency** is the clock frequency of the implemented design. A higher frequency improves the design throughput during execution. A modern HLS tool often employs a parallel-pipelined architectural template to achieve these optimization goals.

Figure 2 shows a common architectural template for designing an FPGA-based hardware accelerator (Cong et al. 2018). This template contains a number of duplicated compute units, which are identical hardware modules that can process

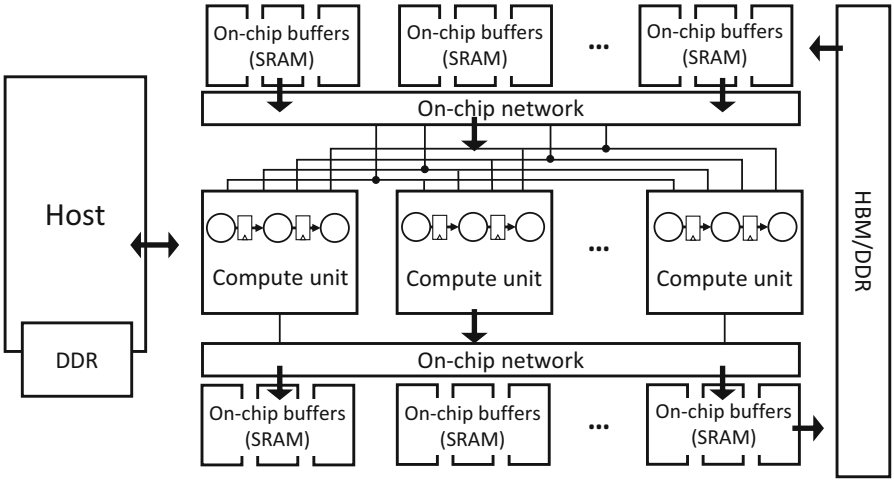


Fig. 2 An architectural template for HLS-based designs

one of the many parallelizable tasks in the design. Such parallel compute units help increase the hardware parallelism. Within each compute unit, there are pipelined compute datapaths and the corresponding control logic. The pipeline can be implemented as coarse-grained task-level pipelines and/or fine-grained loop-level pipelines. These pipelines improve the utilization of the underlying hardware resources. Finally, an application-specific memory hierarchy needs to be built to supply the compute units with enough on-chip and off-chip bandwidth. This includes data reuse to reduce off-chip memory accesses as well as on-chip buffering and partitioning to increase on-chip memory bandwidth. Such a customized memory hierarchy works with the parallelization and pipelining techniques to maximize the throughput of the design.

Modern HLS tools commonly provide a set of compiler transformations and synthesis optimizations to realize such an architectural template. In the following, the authors use this **parallel-pipelined** architectural template with **customized memory hierarchy** to drive the discussion of the rest of this chapter.

Pipelining Techniques

Pipelining is a commonly used technique to improve the throughput of performance-critical regions of the program by temporally overlapping the execution of multiple loop iterations or tasks. By allowing consecutive iterations/tasks to overlap, pipelining improves the utilization of the hardware resources, thus improving the system throughput according to Equation (1). Modern HLS tools support various forms of pipelining that differ in terms of their granularity, their scheduling mechanisms, as well as the composition of the pipelines, which the authors summarize in Table 1.

In this section, the authors first discuss the common optimizations at the operator level. The authors then delve into the details of both statically and dynamically scheduled pipelining techniques with different granularities and compositions.

Operator-Level Optimizations

On general-purpose processors, the operations in the source program are compiled into a fixed set of instructions, where each instruction takes at least one clock cycle. In contrast, FPGA HLS tools can flexibly map the operations onto the heterogeneous resources in a more flexible way to improve the resource utilization and/or increase the clock frequency. Depending on the complexity of the operations being mapped, the HLS tool can either pipeline them to improve timing, or schedule multiple dependent operations into a single cycle to reduce latency. Figure 3 illustrates two important operator-level optimizations, which the authors discuss in more detail as follows.

Operator chaining schedules multiple dependent operations into one clock cycle as long as the estimated delay of the resulting combinational logic does not exceed the target cycle time. The delay estimation usually also takes into account the underlying LUT primitives on an FPGA, so that the operations that are efficiently implementable on an FPGA (e.g., bitwise operations) can be aggressively chained

Table 1 Characterization of common scheduling/pipelining techniques

		Description	Examples
Granularity	**Operator**	Chained/pipelined operators	Chaining (Tan et al. 2015), multi-pumping (Canis et al. 2013)
	Loop	Overlap multiple successive loop iterations	Modulo scheduling (Zhang and Liu 2013)
	Task	Pipeline the invocations of compute tasks	Coarse-grained dataflow (Stefanov et al. 2004)
Mechanism	**Statically scheduled**	Centralized FSM with lock-step execution	Modulo scheduling (Zhang and Liu 2013)
	Dynamically scheduled	Controlled by distributed handshakes	Fine-grained dataflow (Josipović et al. 2018)
Composition	**Heterogeneous**	Different stages have different functionalities	Coarse-grained dataflow (Stefanov et al. 2004)
	Homogeneous	All stages perform (nearly) identical function	Systolic array (Wei et al. 2017)

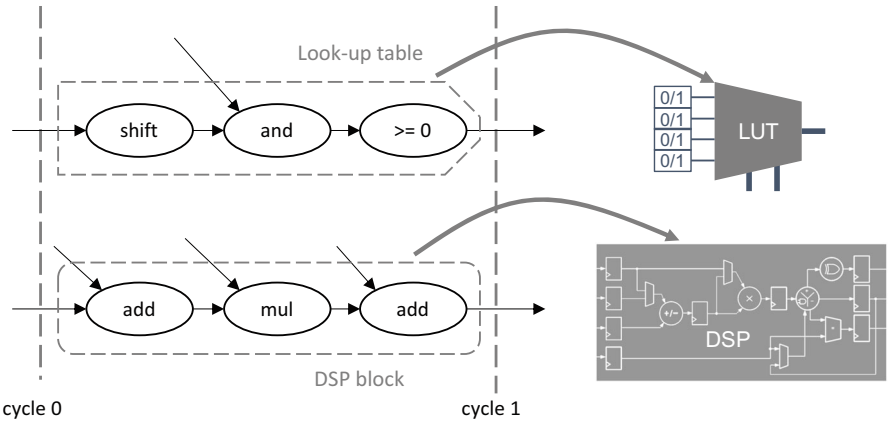


Fig. 3 Operator-level optimizations: operator chaining and DSP mapping

into a single cycle to improve the performance. Operation chaining improves the utilization of the FPGA hardware resources by enabling the execution of multiple operations on a small number of hardware resources. A recent technique considers the underlying LUT mapping optimization during operator chaining to aggressively group operations that can be combined into a single level of LUTs (Tan et al. 2015;

Zhao et al. 2015). The study in Ustun et al. (2020) recognizes the fact that the additive delay model commonly used in the HLS tools does not accurately reflect the true operator-level delay. They propose to use learning-based approaches to predict the operator delay based on features extracted from existing designs as the training set.

Besides operation chaining, modern FPGAs commonly contain dedicated DSP blocks that can implement various common datapath patterns through **DSP mapping**. For example, the Xilinx DSP58 block in the Versal ACAP device includes a 27-bit pre-adder, a 27-by-24-bit multiplier, as well as a 58-bit ALU that can implement different operations such as addition, accumulation, and various bit-level operations (Versal ACAP 2020). In Fig. 3, the add-multiply-add pattern is mapped to the DSP block to utilize the fast pre-adder, multiplier, and post-adder in the DSP block. Effectively detecting such DSP patterns during HLS optimization is critical to achieve high performance. Ronak and Fahmy (2015a) propose an automated three-step approach to partition a dataflow graph into subgraphs that can be mapped to a DSP.

It is also worth noting that the hard blocks such as block memories and DSP units on an FPGA can operate at a higher frequency than the LUT-based soft logic. This provides the HLS tool additional opportunities to improve the resource utilization by clocking such hard blocks at a faster rate. Such techniques are called **multi-pumping** (Canis et al. 2013). For example, when one clocks a DSP block twice as fast as the system clock (i.e., double pumping), one can reuse the same physical block to perform two multiplications in one system clock cycle. The multi-pumping technique has been demonstrated for both the DSP blocks (Ronak and Fahmy 2015b) and the on-chip RAM modules (LaForest and Steffan 2010), showing nontrivial DSP and RAM resource reductions with a small overhead in LUT and register usage.

Statically Scheduled Pipelining

Statically scheduled pipelining refers to the scenario where the overlapping pattern of consecutive loop iterations is statically determined and used during the pipelined execution (Rau and Glaeser 1981). The key performance metric of a statically scheduled pipeline is the initiation interval (II), defined as the time interval (e.g., in number of cycles) between consecutive iterations. The resource utilization of a pipelined loop is inversely proportional to the achievable II. For example, a loop pipeline with an II of 1 can start a new iteration every clock cycle, thus fully utilizing the underlying hardware resources. In general, pipelining time-consuming loops with a small II can significantly improve design throughput according to Equation (1).

The II of a pipeline is fundamentally determined by data dependencies and resource limitations. The HLS scheduler analyzes the dependencies during compile time and schedules the operations such that all dependences and resource constraints are honored. The complexity of constructing a customized pipeline as described in the ▸ Chap. 24, "Accelerator Design with High-Level Synthesis" has led to a rich body of research.

ILP-based approaches (Hwang et al. 1991) are among the early techniques where various scheduling constraints can be expressed as an integer linear program. System of difference constraints (SDC) is used in loop pipelining (Zhang and Liu 2013) to improve the scalability of the ILP-based pipelining formulation by realizing that most of the pipelining constraints are in the form of pairwise difference constraints. The underlying constraint matrix of an SDC system has a special property which guarantees that an integer-valued solution can be efficiently obtained through solving a linear program (Zhang and Liu 2013). Ordering heuristics are proposed to handle constraints that cannot be expressed as linear differences such as the resource constraints. Recently, the authors of Dai et al. (2018) and Dai and Zhang (2019) further extend the SDC-based scheduling formulation by encoding the resource constraints part as a Boolean satisfiability (SAT) problem. The joint SAT-SDC problem is solved iteratively through efficient conflict-driven learning to find the exact solution while achieving a significant speedup over the ILP-based alternatives.

Data hazards in the form of memory dependences limit the achievable II of an HLS pipeline. In Fig. 4, there is a read-after-write (RAW) dependence between two consecutive iterations. Thus, the best achievable II without violating the RAW dependence is 3. However, in many scenarios, the memory dependences only occur

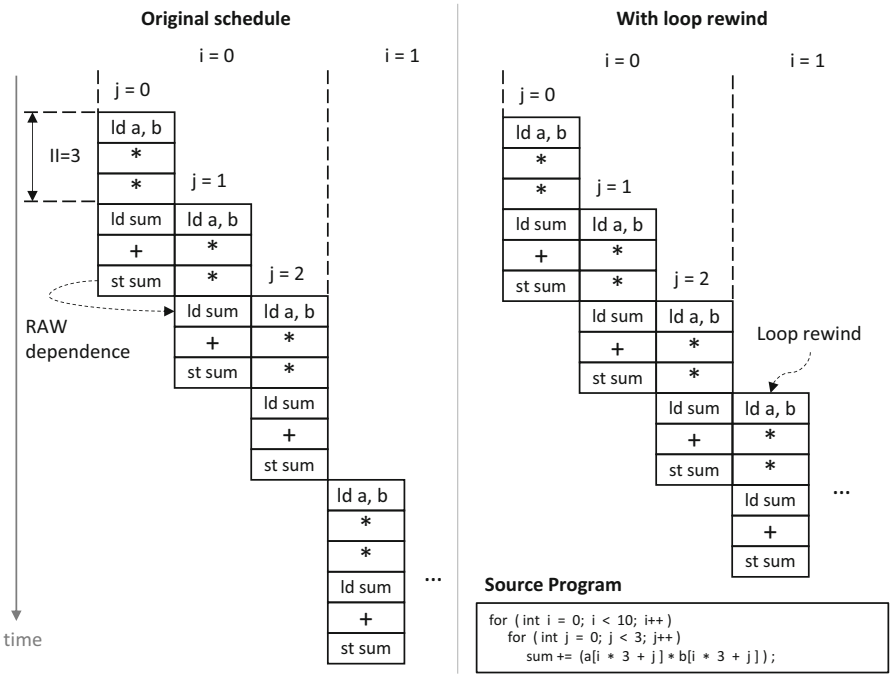


Fig. 4 Statically scheduled pipelining with the RAW data dependence and the loop rewind optimization

infrequently depending on the data-dependent access patterns during execution. Authors of Dai et al. (2017) propose to synthesize dynamic hazard resolution logic into the hardware pipeline, which enables the pipeline to speculatively start new iterations assuming no hazard exists for the particular iteration. In the case where a hazard indeed occurs, the pipeline control logic squashes and replays the current iteration.

Besides inserting hardware logic to dynamically resolve data dependences, there are opportunities during compile time to transform the loop structures to minimize such dependences. Loop interchange is a transformation that swaps the inner and outer level of a multi-level loop. In the case where the original outer loop has a larger dependence distance than the inner loop, it is beneficial to interchange the two levels to relax the dependence distance (Fort et al. 2014). Loop splitting is another loop transformation to split a single loop into multiple sub loops to accommodate for iteration-specific dependencies that can be computed using polyhedral analysis (Liu et al. 2016).

Conventional loop pipelining techniques work only on the innermost loops. For multi-level loop nests, the hardware resources become under-utilized when executing the prologue or the epilogue of the loop. Two techniques have been proposed to improve the utilization for multi-level loop pipelining. Loop flattening uses compiler transformation to convert a perfect two-level loop nest with trip counts M and N into a single-level loop of $M \times N$ iterations (Kato and Kenshu 2013). The flattened loop can then be pipelined without wasting any cycles in between the $M \times N$ iterations. Loop rewind achieves a similar effect of allowing consecutive outer loop iterations to overlap by starting a new invocation of the inner loop before the previous invocation finishes. This removes the bubbles in the pipeline when the inner loop is executed multiple times. Figure 4 shows the effect of loop rewind. In the original schedule without rewinding, the second iteration of the outer loop can only start after its first iteration has completed. With rewinding, the second iteration of the outer loop can start while the first iteration is still executing, thus improving the throughput of the loop nest (See de Fine Licht et al. (2018) for additional loop-level transformations that benefit pipelining.)

Dynamically Scheduled Pipelining

Because the pipeline schedule is fully determined at compile time, statically scheduled pipelining often must assume the worst-case program behavior that result in an overly pessimistic II. This is especially unfavorable for programs that feature many data-dependent control and data dependencies that cannot be accurately analyzed during static compilation (Josipović et al. 2018). Dynamically scheduled pipelining aims to tackle this issue by using a distributed, handshake-based mechanism to control the pipeline execution. Instead of the lockstep execution as in static scheduling, the different stages in a dynamically scheduled pipeline autonomously process the incoming data whenever it becomes available, and produce the output data whenever the downstream stage is ready. This mechanism allows multiple pipeline stages to temporally overlap based on data availability, which further improves resource utilization. The authors categorize the dynamically

scheduled pipelining techniques into two main types. In **heterogeneous pipelining**, each stage of the pipeline generally differs in its functionality. This includes techniques such as coarse- and fine-grained dataflow. In **homogeneous pipelining**, all stages of the pipeline execute exactly the same (or very similar) set of operations. A representative form of homogeneous pipeline are systolic arrays (Kung 1982).

In a **coarse-grained dataflow**, a design is broken into multiple compute processes where the producer-consumer relation between the processes naturally defines a computation graph. The processes communicate data through commonly used hardware communication channels such as FIFOs and ping-pong buffers, where the data are either self-synchronized through the full/empty status of the FIFOs, or synchronized together with the block-level control signals of each process. Theoretical frameworks such as synchronous dataflow (Lee and Messerschmitt 1987) and Kahn process networks (Gilles 1974) provide the foundation for performance analysis and channel sizing of many common classes of the dataflow networks. As shown in Fig. 5, the computation graph can be executed in a pipelined fashion both within a single invocation of the pipeline and across consecutive invocations of the pipeline. Within a single invocation, the consumer process can consume the output data from a producer's FIFO output as soon as new data are present in the FIFO, allowing the consumer to be partially overlapped with the producer in time. Across consecutive invocations, each process in the computation graph can start executing a new task from the next invocation, as soon as it is done with the task in the current invocation and the next input is available.

This dynamically scheduled pipelining technique can also be applied to a **fine-grained dataflow** where each process is either an operation in the source program or a fine-grained dataflow primitive such as a split or join node (Josipovic et al. 2017). Fine-grained dataflow may incur a non-trivial overhead in resource usage due to the extensive handshaking between operations. However, it enables efficient pipelining of irregular loops where the iteration latency can be run-time dependent

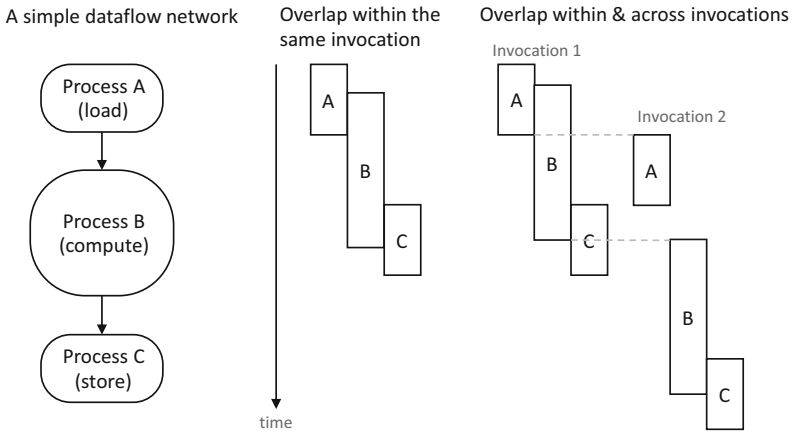


Fig. 5 An example dataflow network and the overlapping of processes

without assuming the worst-case behavior (Josipović et al. 2018). In addition, control and data speculations can readily be supported with additional hardware logic (Josipovic et al. 2019). To reduce the resource overhead, Cheng et al. (2020) propose to combine dynamic and static scheduling by automatically identifying and synthesizing the code regions that can benefit from fine-grained dataflow, while the rest of the design is implemented with static scheduling.

Systolic arrays are composed of a homogeneous network of tightly coupled processing engines (PEs), where each PE only communicates with the neighboring ones. This helps eliminate long interconnects that limit the achievable clock frequency (Zhang et al. 2019). All PEs in a systolic array are often identical (or very similar) in their functionality. For example, a PE in a convolutional neural network engine is composed of a multiply-accumulate compute unit (Wei et al. 2017). Another important feature of a systolic array is the application-specific data reuse logic that helps streaming the input data across the PEs, without needing to store the intermediate results to a long-latency memory (Chi et al. 2018). Several recent efforts have developed automatic synthesis algorithms and programming models to generate high-performance systolic arrays on FPGAs (Cong and Wang 2018; Lai et al. 2020).

Parallelization Techniques

As stated in Equation (1), increasing the parallelism in the design is vital to improve the throughput of an FPGA accelerator. Depending on the design characteristics, an HLS design can often be parallelized by exploiting either data- or task-level parallelism. As illustrated in Fig. 6, vectorization can also be applied at the operator-level to data-parallel applications to widen the datapath. Parallel loops can be unrolled to execute multiple iterations concurrently. In addition, multiple independent tasks can be processed in parallel to exploit task-level parallelism. Figure 6 illustrates the difference between data-level and task-level parallelism. For data-level parallelism, the set of operations performed on different data elements are typically homogeneous. On the other hand, for task-level parallelism, each task can be heterogeneous. In other words, each task executes different jobs in parallel. In the following sections, the authors explain in more detail how the FPGA-specific compilers exploit these different forms of parallelism.

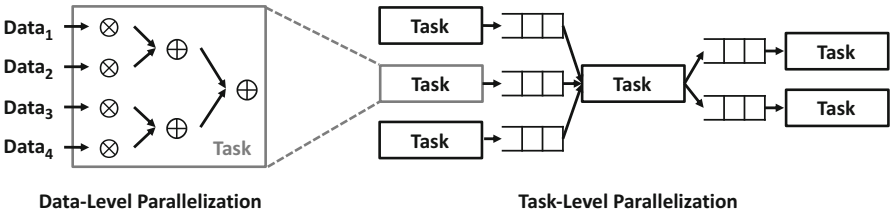


Fig. 6 Illustration of data-level and task-level parallelization

Homogeneous Data-Level Parallelism

Data-level parallelism refers to the property that multiple data elements defined in the source program can be independently processed using the same set of operations. This form of parallelism is typically homogeneous in terms of the compute patterns and can be exploited at different levels of compute granularities through vectorization, unrolling, and multithreading.

Fine-grained data-level parallelism is commonly known as single instruction multiple data (SIMD). In CPUs, such parallelism is exploited using packed SIMD units such as Intel AVX, SSE, and using subword-SIMD extensions. However, since CPUs have to handle exceptions and out-of-order execution, packed SIMD is usually not a scalable solution. Further, these SIMD units only support the native data types (e.g., 8/16/32/64-bit) and thus do not allow variable bitwidth. Modern FPGA HLS compilers also provide support for efficient SIMD computation. The latest Vitis HLS tooland Hipacc (Reiche et al. 2017) allow both automatic SIMD unit generation and user-defined vectorization through the use of vector data types. DSLs such as Halide-HLS (Pu et al. 2017), T2S-Tensor (Srivastava et al. 2019), and SuSy (Lai et al. 2020) allow the user to specify data-level parallelism using the vectorization directives. For example, the matrix multiplication design in T2S-Tensor (Srivastava et al. 2019) has 80 SIMD units, each consisting of sixteen 32-bit single-precision floating-point units.

Loop unrolling is another common technique for exploiting data-level parallelism. In the CPU world, there are many programming frameworks such as OpenMP, Intel Thread Building Block (TBB), and ARM Keil that allow users to unroll loops using either pragmas or special APIs. These unrolled iterations are then run on either different cores, different hyper-threads within the same processor, or are time-multiplexed on a single processor/hyper-thread. Loop unrolling on CPUs can achieve high performance; however, each unrolled iteration is still being run on a processor, and hence the amount of unrolling that can be performed is limited. Moreover, in order to honor memory dependencies, one often needs to use locks for synchronization that can seriously limit the performance gains from unrolling. In contrast, an FPGA device offers much more compute resources than a CPU and hence allows unrolling significantly more iterations of a loop. For example, an Intel Arria 10 FPGA has around 1.5K DSP blocks, each capable of doing one multiply and one add operation on a 32-bit floating point data type. Apart from providing more compute resources, FPGAs also reduce the synchronization overheads between the unrolled iterations by implementing specialized crossbars. FPGA HLS compilers like Xilinx Vivado HLSand Intel FPGA SDK for OpenCLprovide unroll pragmas to allow loop unrolling. Halide-HLS (Pu et al. 2017), T2S-Tensor (Srivastava et al. 2019), and HeteroCL (Lai et al. 2019) allow the user to specify the unrolled loop in high-level functional representation of the computation. AutoAccel (Cong et al. 2018) automatically performs these optimizations on the given source program by analyzing the resource limitations and dependence patterns of the loops. SuSy (Lai et al. 2020) allows the user to specify the computation as uniform recurrent equations (UREs) along with a space-time transformation matrix and automatically applies the unrolled directive to the spatial loops.

In addition to SIMD and unrolling, FPGA accelerators also commonly employ coarse-grained parallelization by running many PEs concurrently in a multithreaded manner. Hsiao et al. propose a technique called thread weaving (Hsiao and Anderson 2019), which statically schedules computation in each thread with the awareness of other threads, and guarantees that the shared resources are accessed in a time-multiplexed and non-interfering manner. FCUDA (Papakonstantinou et al. 2009) can compile CUDA code to FPGAs. This allows programmers to use the high-level CUDA APIs to describe coarse-grain data-level parallelism, where each thread-block inside a CUDA kernel becomes a hardware PE that can be executed in parallel with other PEs. Intel FPGA SDK for OpenCLprovides a GPU-like programming framework to exploit data-level parallelism. It has a triply-nested loop called NDRange. The NDRange is specified using three integer values (R1, R2, R3) which serve as the loop bounds. A single iteration of the loop nest is called a work-item and thus a single kernel execution has $R1 \cdot R2 \cdot R3$ work-items. Such a programming framework naturally exposes data-level parallelism as a single function now operates on the data from $R1 \cdot R2 \cdot R3$ different loops. Software multithreading constructs can be used to explicitly specify coarse-grained parallelism for multiple (often heterogeneous) PEs.

Heterogeneous Task-Level Parallelism

For parallel programming, one can divide the compute into smaller and often dependent tasks that carry out different functions. These heterogeneous tasks can run concurrently and communicate data through message passing or shared memory. On CPUs, since the inter-task communication happens through caches or network, the communication latency is typically high; hence task-level parallelism does not achieve good performance when being performed at a small granularity. In contrast, FPGAs can exploit task level-parallelism at both finer and coarser granularity. This is because of the efficient low-latency on-chip communication that is possible through on-chip interconnect and memories (e.g., registers and BRAMs).

There are various ways of exploiting task-level parallelism in modern HLS compilers. Intel FPGA SDK for OpenCLallows programmers to specify each task as an OpenCL kernel, and different kernels are then connected through channels. These channels can be written and read in both blocking and non-blocking fashion using the corresponding read/write channel function calls. Xilinx Vivado HLSallows the user to specify a directive to infer dataflow-style task pipelining where different functions called within the function/loop are considered as tasks. The communication channels (e.g., FIFOs, double buffers) are automatically connected to the dependent tasks. LegUp allows the user to write multi-threaded software code to express parallelism. The tool automatically synthesizes the parallel-operating hardware sub-circuits and the corresponding synchronization constructs (Choi et al. 2013).

Task-level parallelism is also implemented by many DSLs today. For example, Halide-HLS (Pu et al. 2017) and Darkroom (Hegarty et al. 2014) combine multiple stages of an image processing pipeline and execute them in parallel. On-chip buffers

are used to satisfy the data dependencies. Hipacc (Reiche et al. 2017) uses the concept of streaming objects to represent tasks and uses Vivado HLS streams and Intel FPGA SDK for OpenCL channel interfaces to connect tasks with data dependencies. T2S-Tensor (Srivastava et al. 2019) uses isolate directives to split a task into multiple smaller sub-tasks that are connected via channels and are executed in parallel. TAPA (Chi et al. 2020) is an HLS C++ language extension that enhances the productivity of programming task-parallel applications on FPGAs.

Memory Customization Techniques

The accesses to off-chip memory have higher latency and lower bandwidth compared to the memory accesses to the on-chip memory. The high latency of memory accesses can result in the compute pipeline being stalled for a significant period of time leading to low performance with poor resource utilization. The low off-chip memory bandwidth can result in low parallelism since the compute resources cannot be scaled until there is sufficient bandwidth to supply the data. Thus, achieving low-latency and high-bandwidth memory accesses is essential to achieve high parallelism and high compute utilization, which are the key factors for performance in Equation (1). In this section, the authors first discuss the data reuse buffers and decoupled access-execute architectures that improve the utilization of the compute resources by reducing the memory access latency. The authors then talk about two common approaches for increasing the memory bandwidth: (a) data vectorization and (b) memory banking.

Exploiting Data Reuse

Unlike CPUs, most FPGAs do not offer general-purpose caches that can automatically manage the data reuse. To achieve a high efficiency, FPGA programmers typically construct user-managed buffers using on-chip block RAMs, which can be configured and specialized for a given application. Oftentimes it is not feasible to store the entire active data set in on-chip memories that have limited capacity. In such scenarios, data tiling is used to store a portion of the data on-chip. For data tiling, the input and output data is tiled and cached in the on-chip buffers and the computation is performed on one tile at a time. Note that data tiling is analogous to tiling in CPU programming where the tiling is performed such that the tiles can fit in the on-chip cache. However, tiling in CPU does not have to be exact. In other words, an application for which the input tiles are slightly larger than the cache would still compile and run on CPU due to software-transparent nature of the caches. However, for FPGAs since the on-chip memory is exposed to the programmer, the programmer needs to ensure that a data tile will fit in the on-chip buffers.

For many applications in image processing and machine learning, input accesses are affine expressions of the loop variables. Hence, loop splitting and loop interchange are common techniques to achieve data tiling. For commercial HLS tools

such as Xilinx Vivado HLSand Intel FPGA SDK for OpenCL, loop splitting and data tiling need to be performed manually at the source level. Many DSLs such as Halide-HLS (Pu et al. 2017), T2S-Tensor (Srivastava et al. 2019), and HeteroCL (Lai et al. 2019) allow the user to specify loop splitting and data tiling using loop transformation primitives that are decoupled from the algorithmic specification. Apart from user-specified data tiling, there have been multiple efforts to automatically tile the application data using the polyhedral framework. Chugh et al. (2016) built a DSL on top of PolyMage that tries to maximally exploit the data reuse under the constraints of available on-chip memory capacity and off-chip memory bandwidth. Pouchet et al. (2013) presented an end-to-end system using the polyhedral model which automatically transforms the program for effective data reuse, including the handling of on-chip buffers for FPGAs.

Depending on the memory access patterns, data reuse buffers can be implemented in various forms such as random access buffers, FIFOs, cyclic shift-registers, window buffers, and line buffers. Random access buffers allow data to be read and written at any position. However, these types of buffers do not scale well since the access time for a buffer increases with its size. Memory banking, as is discussed in the next subsection, is a common way to split these large buffers into multiple small buffers. FIFOs are used to provide asynchronous data transfer from producer to consumer in designs that exhibit task-level pipelining. Cyclic shift registers are used for cyclic accesses to a fixed amount of data. Line buffers and window buffers are specific types of buffer implementations that are primarily used in image-processing kernels with sliding window access patterns. The authors show an example of using a pair of line buffer and window buffer in Fig. 7, where one computes a two-dimensional convolution with filter size 3×3. The main purpose of such reuse buffers is to reduce the memory accesses by caching reusable data. For instance, suppose one unrolls loops `r` and `c`; one needs to access input `in` nine times per iteration before applying reuse buffers (Fig. 7a Line 6). After applying reuse buffers, one only needs one access (Fig. 7b Line 12). More details are discussed in the case study (section "Case Study: Binarized Convolutional Neural Networks").

There are different ways of specifying reuse buffers in modern HLS compilers. Xilinx Vivado HLS and Intel FPGA SDK for OpenCL allow the user to implement buffers using the arrays in C/C++/OpenCL. T2S-Tensor (Srivastava et al. 2019) and SuSy (Lai et al. 2020) allow users to insert buffers into a tensor computation using the loop removal and buffer insertion directives. Polymage (Mullapudi et al. 2015) automatically inserts buffers in the generated code for the output of each intermediate function computation in the compute pipeline. HeteroCL (Lai et al. 2019) uses the reuse_at directive to create a reuse buffer. Halide-HLS (Pu et al. 2017), Darkroom (Hegarty et al. 2014), and Hipacc (Reiche et al. 2017) implement mechanisms to specify the line buffer insertion in image processing pipelines.

Decoupled Access-Execute

The concept of decoupled access-execute (DAE) architecture was first introduced by James Smith (1982) in the context of CPUs to hide the memory access latency. Most of the FPGA accelerators today make use of the DAE scheme. Instead of

(a)

```
1 for (y = 0; y < S; y++)
2   for (x = 0; x < S; x++)
3     unrolled for (r = 0; r < 3; r++)
4       unrolled for (c = 0; c < 3; c++)
5         out[y][x] +=
6           in[y+r][x+c] * w[r][c];
```

(b)

```
1  int lb[2][S+2];  // line buffer
2  int wb[3][3];    // window buffer
3  for (y = 0; y < S; y++)
4    for (x = 0; x < S; x++)
5      // shift columns of window buffer
6      unrolled for (i = 0; i < 3; i++)
7        wb[i][0] = wb[i][1];
8        wb[i][1] = wb[i][2];
9      // update window and line buffer
10     wb[0][2] = (lb[0][x]);
11     wb[1][2] = (lb[0][x] = lb[1][x]);
12     wb[2][2] = (lb[1][x] = in[y][x]);
13     unrolled for (r = 0; r < 3; r++)
14       unrolled for (c = 0; c < 3; c++)
15         out[y][x] += wb[r][c] * w[r][c];
```

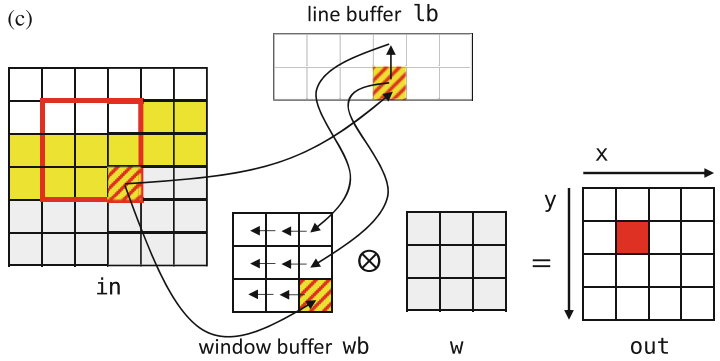


Fig. 7 Example of exploiting data reuse with a pair of window and line buffers. (**a**) HLS code for 2D convolution. (**b**) HLS code after introducing reuse buffers. (**c**) Mechanism of the line buffer and window buffer

having the compute pipeline directly request data from memory, separate data access (read or write) pipelines are instantiated to handle the data movement between the main memory and the accelerator. Since the data access pipeline writes data to on-chip buffers and a compute pipeline reads data from these buffers, it can introduce new stalls in the pipeline. This inefficiency can be solved with double buffers (also known as ping-pong buffers). A double buffer consists of two buffers, one for read, called read buffer, and another one for write, called write buffer. The compute pipeline processes the data tile in the read buffer while at the same time the memory load/store pipeline replaces the data from an old tile with the new tile in the write buffer. When the computation on the read buffer is complete and the write buffer is filled with data, the two buffers are swapped (namely, read becomes write and write becomes read). Double buffering has 2× area overhead as it requires 2× on-chip storage. However, the area overhead is often outweighed by the pipeline efficiency achieved in terms of less stalls and high throughput. The double buffer technique is a form of **latency hiding** technique where one hides the memory latency of loads/stores by overlapping the compute and memory pipelines and thus improve

the utilization in Equation (1). ▶ Chapter 24, "Accelerator Design with High-Level Synthesis" also provided some other memory latency hiding techniques such as hardware-managed caches and prefetchers.

C-based HLS compilers such as Vivado HLS, Intel FPGA SDK for OpenCLand LegUp (Canis et al. 2011) allow specifying double buffers as memory arrays and a boolean variable that determines the read and write buffers. T2S-Tensor (Srivastava et al. 2019) allows the user to specify double buffer as a spatial optimization.

Data Vectorization

Data vectorization helps achieve high off-chip memory bandwidth utilization, which is essential for memory-bound applications where the compute resources cannot be scaled until there is enough memory bandwidth to feed the parallel compute units. With data vectorization, instead of reading/writing a single element, one reads and writes a vector of elements, such as sixteen 32-bit floating point numbers, in the same step. Let us consider the example of a simple DDR model with a bus width of 64 bits and burst length of 8. Whenever a DRAM access is completed and an entire DRAM line is fetched from the memory array, the contiguous data can be sent out to the memory in chunks of 64 bits for up to 8 consecutive cycles. This means that accessing a 16-element vector of 32-bit floating point values achieves higher bandwidth than individually accessing these 16 elements. Thus, data vectorization helps achieve higher compute parallelism which directly results in a higher throughput as in Equation (1).

T2S-Tensor (Srivastava et al. 2019) and Halide-HLS (Pu et al. 2017) allow the user to specify data vectorization using vectorization directives. Intel FPGA SDK for OpenCLprovides vector data-types for integers and floating point numbers that can be used to perform vectorized memory accesses. Depending on the memory access pattern it compiles global memory access into either a burst-coalesced load store unit (LSU) that buffers requests until the largest possible burst can be made, a prefetching LSU that prefetches the data assuming contiguous reads or a cached burst-coalesced LSU which is a burst-coalesced LSU with an on-chip cache. Vivado HLSalso allows the user to pack multiple data elements in a C struct to allow for wide memory accesses. It allows burst-mode data transfer using either a memcpy function in C or a pipelined for loop that accesses memory in a sequential order and where the memory accesses are not placed inside conditional statements.

Memory Banking

An FPGA-based accelerator is typically highly parallelized and/or deeply pipelined in order to achieve a desirable throughput as shown in Fig. 2. As a result, multiple parallel accesses to a single on-chip memory are often required to provide the necessary data bandwidth to sustain the high throughput of the accelerator. However, the embedded memory blocks available on modern FPGA devices (e.g., BRAMs) only provide a very limited number of ports for concurrent reads/writes. Simply replicating the BRAMs to create multi-ported memories is not scalable or even feasible due to the steep area overhead and potential memory coherence overhead resulting from write operations. A more viable solution is memory banking, which partitions a memory block into several smaller banks; thus, concurrent

memory accesses are distributed to different banks, avoiding or minimizing banking conflicts. Since each memory bank only holds a subset of the original memory contents, banking usually yields a significantly lower storage overhead compared to duplication. Nevertheless, additional banking logic is still required to orchestrate the data movement between banked memories and compute units in the accelerator. For non-expert FPGA designers, devising a minimum-conflict banking scheme with low hardware overheads is certainly a challenging task. While commercial HLS tools provide some basic support for array partitioning (Cong et al. 2011), the users remain responsible for manually specifying the banking scheme via vendor-specific pragmas or directives. For this reason, there is an active body of HLS research tackling the problem of automatic array partitioning (i.e., memory banking without access conflicts) given a throughput constraint that is usually specified in terms of pipeline II (Cilardo and Gallo 2015; Meng et al. 2015). More recently, Zhou et al. (2017) propose a trace-driven address mining algorithm that can automatically generate efficient memory banking schemes by analyzing a stream of memory address bits.

Data Type Customization Techniques

Unlike most other general-purpose architectures (There are configurable and extensible processor technologies such as Cadence Tensilica Xtensa (Gonzalez 2000) and Synopsys ASIP Designerthat allow application-specific datapath bitwidth customization.), FPGA has the ability to implement a custom datapath consisting of arithmetic and memory units that are not uniformly sized to a fixed bitwidth. This allows programmers to exploit different numerical data types with the precision tailored for a given application. Such flexibility can substantially improve the efficiency for both compute engines and the custom memory hierarchy. For example, multiple low-bitwidth data elements can be packed together into a wide bit vector without increasing the footprint on the main memory. The packed data can then be read/written in a single memory transaction, which greatly improves the bandwidth utilization and the overall operational intensity of the accelerator. In addition, operations with reduced bitwidth require fewer resources, and thus more compute units can potentially be allocated on the same FPGA device to increase hardware parallelism.

Automatic Bitwidth Optimization

Modern HLS tools commonly provide parameterized integer and fixed-point types through predefined classes or libraries. Besides setting the bitwidth, these data types typically allow the user to configure various other properties such as rounding modes and overflow behaviors. In addition, as detailed in the ▶ Chap. 24, "Accelerator Design with High-Level Synthesis", HLS tools commonly perform automatic bitwidth optimization by propagating the bitwidths of the primary inputs and outputs across the control dataflow graph (CDFG) of the design to minimize the bitwidths of the operators. There is a rich body of literature on the automatic bitwidth or

value range analysis. The common approach is to iteratively propagate the bitwidth information on the underlying dataflow graph using both forward and backward propagations until a fixed point is reached or the gain is diminishing (Stephenson et al. 2000).

Custom Precision Floating-Point Data Types

FPGAs are also suitable for implementing custom precision floating-point arithmetic that is not strictly compliant with the IEEE 754 standard. Several libraries and tools that generate custom FPUs with reduced bitwidth exist (Bansal et al. 2018; Jaiswal and Cheung 2013). For instance, FloPoCo is an open-source C++ framework that can generate customized FPUs in synthesizable VHDL (Dinechin and Pasca 2011). Bansal et al. (2018) propose to use the LegUp HLS tool to implement various custom precision floating-point operators, achieving similar quality of results than hand-optimized RTL implementations. There is an active line of research that explores new floating-point formats to accelerate machine learning workloads. Brain floating point (bfloat) (originally proposed by Google) (Wang and Kanwar 2019) is a truncated version of the IEEE single-precision floating-point format, which is now supported by the Intel Agilex FPGAs. In addition, multiple recent efforts have implemented Posit arithmetic operators on FPGAs (Carmichael et al. 2019; Sommer et al. 2020). The scientific computing domain also sees an increasing adoption of FPGA-based acceleration, partly due to the convenience of implementing floating-point computations that require a very high precision beyond what is being supported by the standard double or even quadruple-precision formats (Govindu et al. 2005; Smith et al. 2005).

Float to Fixed-Point Conversion

While floating-point arithmetic provides a higher dynamic range, it is usually more expensive to implement than their fixed-point counterparts. Float-to-fixed conversion aims to automatically quantize floating-point variables and the associated operations into fixed-point representations to reduce the hardware cost without significantly degrading the accuracy. The existing float-to-fixed conversion techniques can be classified into two major categories in float-to-fixed conversion (Menard et al. 2006): (1) Profiling-driven approaches heuristically determine the bitwidth of the fixed point types, and simulate the source program with representative inputs to measure the performance gain and accuracy loss (Aamodt and Chow 2008). Such approaches can exploit input-dependent opportunities, but may lead to inaccurate results when the profiling is biased. Moreover, the simulation process can potentially be time consuming when the problem size is large. (2) Analytical approaches statically analyze the source program at compile time and estimate the required precision of the fixed-point operations that can meet the accuracy constraint (Cherubin et al. 2019) . Such approaches can provide some theoretical guarantees for relatively simple programs (e.g., DSP kernels) without complex input-dependent control flows. One key challenge is that the static range and precision analysis usually assume worst-case program behaviors, which may yield overly conservative results.

Case Study: Binarized Convolutional Neural Networks

To demonstrate the effectiveness of some of the aforementioned optimizations, the authors present a case study on binary neural networks (BNNs). BNNs represent a promising approach to improving the efficiency of deep learning execution (Courbariaux et al. 2016; Rastegari et al. 2016), especially for convolutional neural networks (CNNs). In a BNN model, (most of) the learnable parameters and feature maps are quantized to $+1/-1$. With binarized weights and activations, the dominant computations of a BNN are binary multiply-accumulate (BMAC) operations, which can be implemented in a highly hardware-efficient way using XNORs and population counts (popcount). The extreme quantization can also reduce the memory requirement for storing the model. As a result, FPGAs are a great match for implementing the BNN models, as the BMAC operations can be mapped and executed on the LUT-based logic fabric in a massively parallel fashion. The reduced memory footprint is also attractive since FPGAs tend to have limited on-chip SRAM capacity. For these reasons, extensive studies have been devoted to the FPGA acceleration of BNNs (Zhao et al. 2017; Zhang et al. 2021).

In this case study, the authors use a simplified BNN model and only focus on the inference process of a single convolutional (conv) layer. In the following, the authors first describe the core convolution algorithm. The authors then discuss how to improve the performance of an BNN accelerator by applying a sequence of HLS optimizations including (1) pipelining and unrolling, (2) custom reuse buffers, and (3) data packing. The authors further show how one can leverage emerging accelerator languages such as HeteroCL to improve the portability and productivity of a program targeting FPGAs. Finally, the authors evaluate each optimization quantitatively by showing the resource and latency numbers from HLS reports.

Algorithm Overview

In a BNN (and CNN in general), a conv layer takes in M input feature maps of size $I \times I$ pixels, convolves them with filters of size $K \times K$ pixels, and produces N output feature maps of size $S \times S$ pixels. The corresponding compute can be expressed as the following equation.

$$out_n(x, y) = \sum_{m=0}^{M-1} \sum_{r=0}^{K-1} \sum_{c=0}^{K-1} in_m(x + c, y + r) \times w_{m,n}(c, r),$$

where $out_n(x, y)$ denotes the value of pixel (x, y) in the nth output feature map, in_m is the mth input feature map, and $w_{m,n}$ is the filter that convolves with input in_m and produces a partial sum of output out_n. Figure 8 shows how a conv layer can be described in a C loop nest.

The main advantage of binarization is that one can replace the expensive multiplications by the cheap bitwise logic operations. Figure 9 shows how one

```
1 for (int m = 0; m < M; m++)            // loop over input feature maps
2   for (int n = 0; n < N; n++)          // loop over output feature maps
3     for (int x = 0; x < S; x++)        // loop over output pixels
4       for (int y = 0; y < S; y++)      // loop over output pixels
5         for (int r = 0; r < K; r++)    // loop over filter pixels
6           for (int c = 0; c < K; c++)  // loop over filter pixels
7             out[n, x, y] += in[m, x+c, y+r] * w[m, n, c, r];
```

Fig. 8 Code snippet for a convolutional layer

(a)

x	y	$x \times y$
-1	-1	$+1$
-1	$+1$	-1
$+1$	-1	-1
$+1$	$+1$	$+1$

(b)

$\hat{x}$	$\hat{y}$	$\hat{x} \odot \hat{y}$
0	0	1
0	1	0
1	0	0
1	1	1

Fig. 9 The encoding and multiplication for binarized variables. (**a**) Normal multiplication between binarized variables x and y. (**b**) Binary multiplication using XNOR with encoded variables $\hat{x}$ and $\hat{y}$

computes a multiplication using an XNOR operation by encoding $+1$ with 1 and -1 with 0. Here $\hat{x}$ denotes the encoded value of x. The dot product between two encoded vectors can be performed as follows:

$$\mathbf{A} \cdot \mathbf{B} = \sum_{i=0}^{L} A_i \times B_i = 2 \sum_{i=0}^{L} (\hat{A}_i \odot \hat{B}_i) - L, \tag{2}$$

Here $\mathbf{A}$ and $\mathbf{B}$ are two vectors with the same length L (i.e., $|\mathbf{A}| = |\mathbf{B}| = L$), A_i and B_i are binarized values that are either $+1$ or -1, and $\hat{A}_i$ and $\hat{B}_i$ are encoded values for A_i and B_i according to Fig. 9. The summation ($\sum$) in Equation (2) requires an integer addition. A more concrete example is shown below, where the entire dot product is multiplierless.

$$[+1, -1, -1, +1] \cdot [-1, -1, +1, +1] = 1 \times -1 + -1 \times -1 + -1 \times 1 + 1 \times 1 = 0$$
$$= 2 \times (1 \odot 0 + 0 \odot 0 + 0 \odot 1 + 1 \odot 1) - 4 = 0$$

Pipelining and Unrolling

To achieve a high performance, loop pipelining (section "Pipelining Techniques") and unrolling (section "Parallelization Techniques") are key optimizations. However, given the loop nest in Fig. 8, one has many choices for parallelization. Figure 10 shows an example of different sources of parallelism within a convolution layer. Exploiting the parallelism from different sources introduces trade-offs between performance and resource usage. Ideally, one would unroll the entire loop

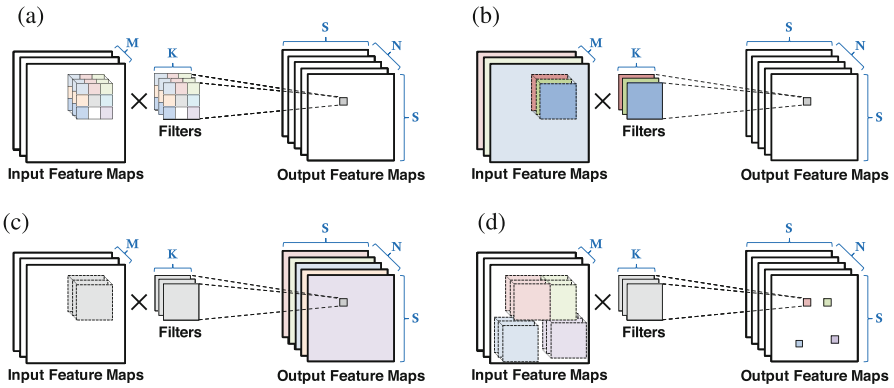


Fig. 10 Sources of parallelism within a convolution layer. (**a**) Parallelism across filter pixels. (**b**) Parallelism across input feature maps. (**c**) Parallelism across output feature maps. (**d**) Parallelism across output pixels

```
1  pipelined for (int m_n_x_y = 0; m_n_x_y < M*N*S*S; m_n_x_y++)
2    // recovering indices from the fused loop
3    y = m_n_x_y % S;
4    x = (m_n_x_y / S) % S;
5    n = (m_n_x_y / S*S) % N;
6    m = m_n_x_y / S*S*N;
7    int sum = 0;
8    unrolled for (int r = 0; r < K; r++)
9      unrolled for (int c = 0; c < K; c++)
10       sum += (in[m, x+c, y+r] == w[m, n, c, r]) << 1 - 1;
11   out[n, x, y] += sum;
```

Fig. 11 BNN code snippet with loop pipelining, unrolling, and fusion

nest to achieve the highest performance. However, such an approach is usually impractical considering the complexity of the loop nest and the available on-chip resources. A common practice of optimizing BNN (and CNN in general) is to exploit the parallelism across filter pixels (Fig. 10a) since the filter size (i.e., $K \times K$) is rather small. One can further combine other sources of parallelism if there are enough resources. In this section, the authors only focus on the parallelism across filter pixels. The authors show how to combine other sources of parallelism in the later sections with optimizations such as data type optimizations. Figure 11 shows the code snippet after unrolling the loops over the filter pixels (i.e., `r` and `c`) and pipelining the outer loops. The authors also apply loop fusion (section "Statically Scheduled Pipelining") to increase the pipeline efficiency.

However, if one simply pushes this code through HLS synthesis, the scheduling tools would fail to achieve II = 1. The main reason is the resource contention caused by the on-chip memory accesses. More concretely, after loop unrolling, there are $K \times K$ accesses to `in` in a single cycle, while on FPGAs the typical number of ports per SRAM block is two. Thus, if $K = 3$, the maximum achievable II is $\lceil 9/2 \rceil = 5$.

A similar issue also exists for `w`. To overcome the port limitation, one needs to perform memory optimizations, which the authors discuss in the next section.

Line Buffers and Window Buffers

As explained in section "Memory Customization Techniques", data reuse and memory banking are two key optimizations for resolving resource contention. In this case study, to resolve the resource contention for `w`, one can simply apply memory banking (section "Memory Banking") by fully partitioning `w`. However, for `in`, it is not practical to fully partition it because of the size. Instead, the authors create reuse buffers (section "Exploiting Data Reuse") to store the pixels that are accessed repeatedly in consecutive iterations. For instance, if $K = 3$, an input pixel is reused $K \times K = 9$ times.

In this case study, the authors create a line buffer and a window buffer as shown in Fig. 7c. The main purpose of the line buffer is to cache the reusable input pixels (i.e., the pixels highlighted in yellow) so that one can reduce the memory accesses from the input feature map. As for the window buffer, the purpose is to cache the input pixels for performing the convolution (i.e., the pixels in the red box). Thus, the size of a window buffer is the same as that of a filter. In order to realize the parallelism across filter pixels, the authors implement the window buffer with flip flops such that $K \times K$ dot products can be performed in parallel.

In the following the authors briefly describe how the line buffer and window buffer operate. In each iteration, the window buffer is convolved with the filters `w` to generate one output value. In the next iteration, the window buffer is updated by shifting the columns to the left, reading two values from the line buffer, and reading one value from the input `in`. Meanwhile, the line buffer is also updated by shifting up one column and reading one input value. In this scenario, only one input pixel is accessed per iteration instead of nine pixels. Namely, II = 1 can be achieved.

Data Vectorization

The next optimization one can perform is data vectorization, which packs the binarized values into long-bitwidth integers. The vectorized data can then be read/written in a single memory transaction, which greatly improves the bandwidth utilization and the overall operational intensity. To realize data vectorization in HLS, arbitrary-precision data types are essential to describe the packed data (e.g., `ap_(u)int` for Xilinx HLS and `ac_(u)int` for Intel HLS).

Similar to pipelining and unrolling, data vectorization exploits another source of parallelism in a conv layer. In this case study, the authors choose to vectorize along the input feature maps (i.e., Fig. 10b) since it works well with our pipelining and unrolling scheme. After vectorization, the dot products become popcount operations. Figure 12 shows the algorithm after applying bit packing. Note that after vectorization, loop `m` does not exist within the fused pipelined loop anymore.

```
1 pipelined for (int n_x_y = 0; n_x_y < N*S*S; n_x_y++)
2   // recovering indices from the fused loop
3   ...
4   int sum = 0;
5   unrolled for (int r = 0; r < K; r++)
6     unrolled for (int c = 0; c < K; c++)
7       // inv and wv are the vectorized input feature maps and weights
8       sum += popcount(inv[x+c, y+r], wv[n, c, r]) << 1 - M;
9   out[n, x, y] = sum;
```

Fig. 12 BNN code snippet after applying data vectorization

Building the BNN Accelerator Using HeteroCL

As discussed in section "Domain-Specific Languages (DSLs)", emerging accelerator design languages can raise the abstraction level of HLS, which allows programmers to focus on high-level design and optimization decisions. In this case study, the authors use HeteroCL as a representative accelerator design language and compare it with HLS C/C++. HeteroCL is a Python-based programming framework that blends declarative symbolic expressions with imperative code. Furthermore, it adopts the idea of separation of concerns, where the algorithm description is separated from the specification of hardware customizations.

Figure 13a shows an example of using HeteroCL to describe the optimized binarized convolutional layer, and Fig. 13b shows the corresponding HLS code. In the algorithm specification, the authors use HeteroCL APIs such as `hcl.compute` and `hcl.sum` to specify the main computation (Lines 7–11). For the optimizations, HeteroCL supports a rich set of customization techniques in a decoupled fashion (Lines 14–17). First, for pipelining and parallelization, the authors use `s.pipeline` to specify the loop to be pipelined, which corresponds to the insertion of a vendor-specific directive in Line 9 of Fig. 13b. Second, for memory customization, the authors use `s.reuse_at` to insert reuse buffers (Lines 15–16), which corresponds to the line buffer and window buffer operations in Lines 10–20 of Fig. 13b. Finally, for data type customization, HeteroCL supports arbitrary-precision integers such as `hcl.UInt(1)` for a one-bit unsigned integer (Lines 2–3). The authors also use APIs such as `hcl.pack` to specify data packing (Lines 5–6), which corresponds to the packed inputs in Line 1 of Fig. 13b. HeteroCL automatically infers the new data type and tensor shapes after packing. Compared to the HLS code, the HeteroCL program is more compact and potentially achieves higher productivity and portability with decoupled algorithm specification and hardware customizations.

Evaluation

To demonstrate the effectiveness of the optimizations, the authors conduct a set of experiments with HLS code generated from HeteroCL and show the resource usage as well as the latency in terms of the number of clock cycles after applying

(a)

```
import heterocl as hcl
in = hcl.placeholder((M, S+K-1, S+K-1), dtype=hcl.UInt(1))
w = hcl.placeholder((M, N, K, K), dtype=hcl.UInt(1))
def bin_conv(in, w):
    inv = hcl.pack(in, factor=M)
    wv = hcl.pack(w, factor=M)
    r = hcl.reduce_axis(0, K)
    c = hcl.reduce_axis(0, K)
    out = hcl.compute((N, S, S), lambda n, x, y: hcl.sum(
        (hcl.popcount(inv[x+r, y+c]^wv[n, r, c]) << 1) - 16,
        axis=[r, c]))
    return out

s = hcl.create_schedule([in, w], bin_conv)
LB = s.reuse_at(inv, s[out], out.axis[1])
WB = s.reuse_at(LB, s[out], out.axis[2])
s[out].pipeline(out.axis[2])
```

(b)

```
void bin_conv(ap_uint<16> inv[S+K-1][S+K-1], ap_uint<16> wv[N][K][K],
              int out[N][S][S]) {
  ap_uint<16> LB[K][S+K-1], WB[K][K];
  #pragma HLS array_partition variable=LB dim=2
  #pragma HLS array_partition variable=WB
  for (int n = 0; n < N; ++n) {
    for (int x_reuse = 0; x_reuse < S+K-1; ++x_reuse) {
      for (int y_reuse = 0; y_reuse < S+K-1; ++y_reuse) {
      #pragma HLS pipeline
        for (int inv_1 = 0; inv_1 < K-1; ++inv_1) {
          LB[inv_1][y_reuse] = LB[(inv_1 + 1)][y_reuse];
        }
        LB[K-1][y_reuse] = inv[x_reuse][y_reuse];
        if (K-1 <= x_reuse) {
          for (int LB_1 = 0; LB_1 < K; ++LB_1) {
            for (int LB_0 = 0; LB_0 < K-1; ++LB_0) {
              WB[LB_1][LB_0] = WB[LB_1][(LB_0 + 1)];
            }
            WB[LB_1][K-1] = LB[LB_1][y_reuse];
          }
          if (K-1 <= y_reuse) {
            int sum = 0;
            for (int r = 0; r < K; ++r) {
              for (int c = 0; c < K; ++c) {
                sum += (popcount(WB[r][c] ^ wv[n][r][c]) << 1) - M;
            }}
            out[n][(x_reuse - K+1)][(y_reuse - K+1)] = sum;
}}}}}}
```

Fig. 13 Optimized BNN codes in different programming languages. (**a**) HeteroCL. (**b**) HLS

Table 2 BNN accelerator evaluation results

	LUT	FF	DSP	BRAM	Achieved II	Latency	Speedup
Baseline	755	298	0	0	N/A	1,171,073	1.0×
+Pipelining & Unrolling	1,079	342	0	0	5	230,401	5.1×
+Reuse Buffers	1,047	331	1	0	1	61,957	18.9×
+Bit Packing	2,427	596	1	0	1	3,877	302.0×

each optimization cumulatively. For this specific design, the lower latency leads to a higher throughput. In our experiments, the authors select $M = 16$, $N = 32$, $S = 9$, $K = 3$. The target device and clock period are Xilinx Zynq and 10 ns, respectively. Table 2 shows the final results. Comparing to HLS C/C++, the programmer can carry out most optimization steps in HeteroCL by simply applying different customization primitives without modifying the algorithm, which is usually a more productive and less error-prone process.

From the table, one can see that pipelining and unrolling can effectively reduce the latency, while also increasing the resource usage. Moreover, the achieved II is only five because of the resource contention, which becomes the performance bottleneck. To solve that, by using reuse buffers (i.e., line buffers and window buffers), now II = 1 can be achieved, which provides another 5.0× speedup as expected. Finally, with bit packing, the authors achieve another 15.9× speedup since the authors exploit the parallelism across the input feature maps (i.e., the authors pack $M = 16$ bits into a single integer). To sum up, the authors achieve around 592× speedup over the baseline design while using approximately 4× more resources.

Concluding Remarks

This chapter surveys popular FPGA HLS compilers and discusses the key optimizations in these tools to improve the throughput of FPGA-based designs. Such optimizations improve one or more of the performance factors: parallelism, utilization, and clock frequency. The authors describe representative optimization techniques in the literature in four major categories, including pipelining, parallelization, memory customization, and datatype customization. The authors also present a case study on an HLS-based BNN accelerator to show the impacts of these techniques.

It is clear that the latest generation of FPGA-targeted compilers and programming languages have made significant progress in making FPGA more accessible to software developers. As a result, FPGA programmers can now more productively implement various important hardware customization techniques to build an efficient accelerator with high quality of results. Despite this encouraging development, current FPGAs are not fully software programmable – at least not in the conventional manner that works for microprocessors, even with the introduction of HLS. It may still take hours or days to compile HLS-generated

RTL to bitstream due to the slow physical design steps. Clearly, there remains a host of challenges and opportunities for FPGA-specific compilers to compile a high-level software specification to bitstream within minutes, and with significantly less manual effort involved. To this end, it is to our belief that physical-aware HLS (Guo et al. 2020, 2021) and bottom-up modular place-and-route will be crucial to enable a much faster FPGA design closure. The authors also believe that domain-specific overlay architectures (Abdelfattah et al. 2018; Ma et al. 2019) will play an increasingly important role in providing a programming experience that resembles software development. Furthermore, integration with the high-level software stack is necessary in order to increase adoption of FPGAs by software programmers.

References

Aamodt TM, Chow P (2008) Compile-time and instruction-set methods for improving floating-to fixed-point conversion accuracy. In: ACM transactions in embedded computing systems (TECS)

Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M et al (2016) Tensorflow: a system for large-scale machine learning. In: USENIX symposium on operating systems design and implementation (OSDI)

Abdelfattah MS, Han D, Bitar A, DiCecco R, O'Connell S, Shanker N, Chu J, Prins I, Fender J, Ling AC et al (2018) Dla: compiler and FPGA overlay for neural network inference acceleration. In: International conference on field programmable logic and applications (FPL)

Bacon DF, Rabbah R, Shukla S (2013) FPGA programming for the masses. In: Communications of the ACM

Bansal S, Hsiao H, Czajkowski T, Anderson JH (2018) High-level synthesis of software-customizable floating-point cores. In: Design, automation, and test in Europe (DATE)

Bezati E, Emami M, Larus J (2020) Advanced dataflow programming using actor machines for high-level synthesis. In: International symposium on field-programmable gate arrays (FPGA)

Canis A, Choi J, Aldham M, Zhang V, Kammoona A, Anderson JH, Brown S, Czajkowski T (2011) LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In: International symposium on field-programmable gate arrays (FPGA)

Canis A, Anderson JH, Brown SD (2013) Multi-pumping for resource reduction in FPGA high-level synthesis. In: Design, automation, and test in Europe (DATE)

Carlson T, Van Wyk E (2019) Building parallel programming language constructs in the AbleC extensible C compiler framework: a PPoPP tutorial. In: ACM SIGPLAN conference on principles and practice of parallel programming (PPoPP)

Carmichael Z, Langroudi HF, Khazanov C, Lillie J, Gustafson JL, Kudithipudi D (2019) Deep positron: a deep neural network using the posit number system. In: Design, automation, and test in Europe (DATE)

Chen T, Li M, Li Y, Lin M, Wang N, Wang M, Xiao T, Xu B, Zhang C, Zhang Z (2015) Mxnet: a flexible and efficient machine learning library for heterogeneous distributed systems. arXiv preprint arXiv:1512.01274

Chen T, Moreau T, Jiang Z, Zheng L, Yan E, Shen H, Cowan M, Wang L, Hu Y, Ceze L et al (2018) TVM: an automated end-to-end optimizing compiler for deep learning. In: USENIX symposium on operating systems design and implementation (OSDI)

Cheng J, Josipovic L, Constantinides GA, Ienne P, Wickerson J (2020) Combining dynamic & static scheduling in high-level synthesis. In: International symposium on field-programmable gate arrays (FPGA)

Cherubin S, Cattaneo D, Chiari M, Di Bello A, Agosta G (2019) TAFFO: tuning assistant for floating to fixed point optimization. In: IEEE embedded systems letters

Chi Y, Cong J, Wei P, Zhou P (2018) SODA: stencil with optimized dataflow architecture. In: International conference on computer-aided design (ICCAD)
Chi Y, Guo L, Choi Y-K, Wang J, Cong J (2020) Extending high-level synthesis for task-parallel programs. arXiv preprint arXiv:2009.11389
Choi J, Brown S, Anderson J (2013) From software threads to parallel hardware in high-level synthesis for FPGAs. In: International conference on field programmable technology (FPT)
Choi Y-K, Cong J, Fang Z, Hao Y, Reinman G, Wei P (2016) A quantitative analysis on microarchitectures of modern CPU-FPGA platforms. In: Design automation conference (DAC)
Chugh N, Vasista V, Purini S, Bondhugula U (2016) A DSL compiler for accelerating image processing pipelines on FPGAs. In: International conference on parallel architectures and compilation
Cilardo A, Gallo L (2015) Improving multibank memory access parallelism with lattice-based partitioning. In: ACM transactions on architecture and code optimization (TACO)
Cong J, Wang J (2018) PolySA: polyhedral-based systolic array auto-compilation. In: International conference on computer-aided design (ICCAD)
Cong J, Liu B, Neuendorffer S, Noguera J, Vissers K, Zhang Z (2011) High-level synthesis for FPGAs: from prototyping to deployment. In: IEEE transactions on computer-aided design of integrated circuits and systems (TCAD)
Cong J, Wei P, Yu CH, Zhang P (2018) Automated accelerator generation and optimization with composable, parallel and pipeline architecture. In: Design automation conference (DAC)
Courbariaux M, Hubara I, Soudry D, El-Yaniv R, Bengio Y (2016) Binarized neural networks: training deep neural networks with weights and activations constrained to +1 or -1. arXiv preprint arXiv:1602.02830
Coussy P, Chavet C, Bomel P, Heller D, Senn E, Martin E (2008) GAUT: a high-level synthesis tool for DSP applications. Springer, Dordrecht
Dai S, Zhang Z (2019) Improving scalability of exact modulo scheduling with specialized conflict-driven learning. In: Design automation conference (DAC)
Dai S, Zhao R, Liu G, Srinath S, Gupta U, Batten C, Zhang Z (2017) Dynamic hazard resolution for pipelining irregular loops in high-level synthesis. In: International symposium on field-programmable gate arrays (FPGA)
Dai S, Liu G, Zhang Z (2018) A scalable approach to exact resource-constrained scheduling based on a joint SDC and SAT formulation. In: International symposium on field-programmable gate arrays (FPGA)
de Fine Licht J, Besta M, Meierhans S, Hoefler T (2018) Transformations of high-level synthesis codes for high-performance computing. arXiv preprint arXiv:1805.08288
Dennis JB (1974) First version of a data flow procedure language. In: Programming symposium
De Dinechin F, Pasca B (2011) Designing custom arithmetic data paths with FloPoCo. IEEE Des Test Comput 28:18–27
Eker J, Janneck J (2003) CAL language report: specification of the CAL actor language. EECS Department, University of California, Berkeley Technical Report
Fort B, Canis A, Choi J, Calagar N, Lian R, Hadjis S, Chen YT, Hall M, Syrowik B, Czajkowski T et al (2014) Automating the design of processor/accelerator embedded systems with legup high-level synthesis. In: International conference on embedded and ubiquitous computing
Gaide B, Gaitonde D, Ravishankar C, Bauer T (2019) Xilinx adaptive compute acceleration platform: VersalTM architecture. In: International symposium on field-programmable gate arrays (FPGA)
Gilles K (1974) The semantics of a simple language for parallel programming. In: Information processing
Gonzalez RE (2000) Xtensa: a configurable and extensible processor. IEEE Micro 20:60–70
Govindu G, Scrofano R, Prasanna VK (2005) A library of parameterizable floating-point cores for FPGAs and their application to scientific computing. In: International conference on engineering reconfigurable systems and algorithms
Guo L, Lau J, Chi Y, Wang J, Yu CH, Chen Z, Zhang Z, Cong J (2020) Analysis and optimization of the implicit broadcasts in FPGA HLS to improve maximum frequency. In: Design automation conference (DAC)

Guo L, Chi Y, Wang J, Lau J, Qiao W, Ustun E, Zhang Z, Cong J (2021) AutoBridge: coupling coarse-grained floorplanning and pipelining for high-frequency HLS design on multi-die FPGAs. In: International symposium on field-programmable gate arrays (FPGA)

Hadjis S, Olukotun K (2019) Tensorflow to coud FPGAs: tradeoffs for accelerating deep neural networks. In: International conference on field programmable logic and applications (FPL)

Hegarty J, Brunhaver J, DeVito Z, Ragan-Kelley J, Cohen N, Bell S, Vasilyev A, Horowitz M, Hanrahan P (2014) Darkroom: compiling high-level image processing code into hardware pipelines. ACM Trans Graph (TOG) 33:1–11

Hegarty J, Daly R, DeVito Z, Ragan-Kelley J, Horowitz M, Hanrahan P (2016) Rigel: flexible multi-rate image processing hardware. In: ACM transactions on graphics (TOG)

Hsiao H, Anderson J (2019) Thread weaving: static resource scheduling for multithreaded high-level synthesis. In: Design automation conference (DAC)

Hwang C-T, Lee J-H, Hsu Y-C (1991) A formal approach to the scheduling problem in high-level synthesis. In: IEEE transactions on computer-aided design of integrated circuits and systems (TCAD)

Ishikawa A, Fukushima N, Maruoka A, Iizuka T (2019) Halide and GENESIS for generating domain-specific architecture of guided image filtering. In: International symposium on circuits and systems (ISCAS)

Jaiswal MK, Cheung RCC (2013) Area-efficient architectures for double precision multiplier on FPGA, with run-time-reconfigurable dual single precision support. Microelectron J 44:421–430

Janneck JW, Miller ID, Parlour DB, Roquier G, Wipliez M, Raulet M (2011) Synthesizing hardware from dataflow programs. J Sig Process Syst 63:241–249

Jia Y, Shelhamer E, Donahue J, Karayev S, Long J, Girshick R, Guadarrama S, Darrell T (2014) Caffe: convolutional architecture for fast feature embedding. In: International conference on multimedia

Josipovic L, Brisk P, Ienne P (2017) From C to elastic circuits. In: Asilomar conference on signals, systems, and computers

Josipović L, Ghosal R, Ienne P (2018) Dynamically scheduled high-level synthesis. In: International symposium on field-programmable gate arrays (FPGA)

Josipovic L, Guerrieri A, Ienne P (2019) Speculative dataflow circuits. In: International symposium on field-programmable gate arrays (FPGA)

Kato Y, Seto K (2013) Loop fusion with outer loop shifting for high-level synthesis. In: IPSJ transactions on system LSI design methodology

King M, Dave N (2012) Automatic generation of hardware/software interfaces. In: ACM SIGARCH computer architecture news

Koeplinger D, Prabhakar R, Zhang Y, Delimitrou C, Kozyrakis C, Olukotun K (2016) Automatic generation of efficient accelerators for reconfigurable hardware. In: International symposium on computer architecture (ISCA)

Koeplinger D, Feldman M, Prabhakar R, Zhang Y, Hadjis S, Fiszel R, Zhao T, Nardi L, Pedram A, Kozyrakis C et al (2018) Spatial: a language and compiler for application accelerators. In: ACM SIGPLAN conference on programming language design and implementation (PLDI)

Kung H-T (1982) Why systolic architectures? IEEE Comput 15:37–46

LaForest CE, Steffan JG (2010) Efficient multi-ported memories for FPGAs. In: International symposium on field-programmable gate arrays (FPGA)

Lai Y-H, Chi Y, Hu Y, Wang J, Yu CH, Zhou Y, Cong J, Zhang Z (2019) HeteroCL: a multi-paradigm programming infrastructure for software-defined reconfigurable computing. In: International symposium on field-programmable gate arrays (FPGA)

Lai Y-H, Rong H, Zheng S, Zhang W, Cui X, Jia Y, Wang J, Sullivan B, Zhang Z, Liang Y et al (2020) SuSy: a programming model for productive construction of high-performance systolic arrays on FPGAs. In: International conference on computer-aided design (ICCAD)

Lee EA, Messerschmitt DG (1987) Synchronous data flow. In: Proceedings of the IEEE

Lee H, Brown K, Sujeeth A, Chafi H, Rompf T, Odersky M, Olukotun K (2011) Implementing domain-specific languages for heterogeneous parallel computing. In: IEEE Micro

Liao S-W, Kuang S-Y, Kao C-L, Tu C-H (2019) A halide-based synergistic computing framework for heterogeneous systems. J Sig Process Syst 91:219–233
Li J, Chi Y, Cong J (2020) HeteroHalide: from image processing DSL to efficient FPGA acceleration. In: International symposium on field-programmable gate arrays (FPGA)
Lindtjorn O, Clapp R, Pell O, Fu H, Flynn M, Mencer O (2011) Beyond traditional microprocessors for geoscience high-performance computing applications. In: IEEE Micro
Liu J, Wickerson J, Constantinides GA (2016) Loop splitting for efficient pipelining in high-level synthesis. In: IEEE symposium on field programmable custom computing machines (FCCM)
Ma R, Hsu J-C, Tan T, Nurvitadhi E, Sheffield D, Pelt R, Langhammer M, Sim J, Dasu A, Chiou D (2019) Specializing fgpu for persistent deep learning. In: International conference on field programmable logic and applications (FPL)
Meeuws R, Galuzzi C, Bertels K (2011) High level quantitative hardware prediction modeling using statistical methods. In: International conference on embedded computer systems: architectures, modeling and simulation
Menard D, Chillet D, Sentieys O (2006) Floating-to-fixed-point conversion for digital signal processors. EURASIP J Adv Sig Process
Meng C, Yin S, Ouyang P, Liu L, Wei S (2015) Efficient memory partitioning for parallel data access in multidimensional arrays. In: Design automation conference (DAC)
Merlin Compiler (2020) Falcon Computing Solutions. https://github.com/falconcomputing/merlin-compiler
Moreau T, Chen T, Jiang Z, Ceze L, Guestrin C, Krishnamurthy A (2018) VTA: an open hardware-software stack for deep learning. arXiv preprint arXiv:1807.04188
Mullapudi RT, Vasista V, Bondhugula U (2015) Polymage: automatic optimization for image processing pipelines. ACM SIGARCH Comput Archit News 43:429–443
Najjar WA, Villarreal J, Halstead RJ (2016) ROCCC 2.0. In: FPGAs for software programmers
Nane R, Sima VM, Quoc CP, Goncalves F, Bertels K (2014) High-level synthesis in the Delft workbench hardware/software co-design tool-chain. In: International conference on embedded and ubiquitous computing
Nigam R, Atapattu S, Thomas S, Li Z, Bauer T, Ye Y, Koti A, Sampson A, Zhang Z (2020) Predictable accelerator design with time-sensitive affine types. In: ACM SIGPLAN conference on programming language design and implementation (PLDI)
Ostadzadeh SA, Meeuws RJ, Galuzzi C, Bertels K (2010) Quad–a memory access pattern analyser. In: International symposium on applied reconfigurable computing
Papakonstantinou A, Gururaj K, Stratton JA, Chen D, Cong J, Hwu W-MW (2009) FCUDA: enabling efficient compilation of CUDA kernels onto FPGAs. In: Symposium on application specific processors (SASP)
Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin Z, Gimelshein N, Antiga L et al (2019) Pytorch: an imperative style, high-performance deep learning library. arXiv preprint arXiv:1912.01703
Peverelli F, Rabozzi M, Del Sozzo E, Santambrogio MD (2018) OXiGen: a tool for automatic acceleration of C functions into dataflow FPGA-based kernels. In: International parallel and distributed processing symposium on workshop (IPDPSW)
Pilato C, Ferrandi F (2013) Bambu: a modular framework for the high-level synthesis of memory-intensive applications. In: International conference on field programmable logic and applications (FPL)
Pouchet L-N, Zhang P, Sadayappan P, Cong J (2013) Polyhedral-based data reuse optimization for configurable computing. In: International symposium on field-programmable gate arrays (FPGA)
Pu J, Bell S, Yang X, Setter J, Richardson S, Ragan-Kelley J, Horowitz M (2017) Programming heterogeneous systems from an image processing DSL. ACM Trans Archit Code Optim (TACO) 14:1–25
Putnam A, Bennett D, Dellinger E, Mason J, Sundararajan P, Eggers S (2008) CHiMPS: a c-level compilation flow for hybrid CPU-FPGA architectures. In: International conference on field programmable logic and applications (FPL)

Ragan-Kelley J, Barnes C, Adams A, Paris S, Durand F, Amarasinghe S (2013) Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. ACM SIGPLAN Not 48:519–530

Rastegari M, Ordonez V, Redmon J, Farhadi A (2016) Xnor-net: imagenet classification using binary convolutional neural networks. In: European conference on computer vision

Rau BR, Glaeser CD (1981) Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In: ACM SIGMICRO newsletter

Reiche O, Özkan MA, Membarth R, Teich J, Hannig F (2017) Generating FPGA-based image processing accelerators with Hipacc. In: International conference on computer-aided design (ICCAD)

Ronak B, Fahmy SA (2015a) Mapping for maximum performance on FPGA DSP blocks. In: IEEE transactions on computer-aided design of integrated circuits and systems (TCAD)

Ronak B, Fahmy SA (2015b) Minimizing DSP block usage through multi-pumping. In: International conference on field programmable technology (FPT)

Rong H (2017) Programmatic control of a compiler for generating high-performance spatial hardware. arXiv preprint arXiv:1711.07606

Rong H (2018a) Productively expressing high-performance spatial designs of givens rotation-based QR decomposition algorithm. arXiv preprint arXiv:1805.07490

Rong H (2018b) Expressing sparse matrix computations for productive performance on spatial architectures. arXiv preprint arXiv:1810.07517

Sérot J, Berry F, Ahmed S (2013) CAPH: a language for implementing stream-processing applications on FPGAs. In: Embedded systems design with FPGAs

Singh S, Greaves DJ (2008) Kiwi: synthesis of FPGA circuits from parallel programs. In: IEEE symposium on field programmable custom computing machines (FCCM)

Skalicky S, Monson J, Schmidt A, French M (2018) Hot & spicy: improving productivity with python and HLS for FPGAs. In: IEEE symposium on field programmable custom computing machines (FCCM)

Smith JE (1982) Decoupled access/execute computer architectures. Comput Archit News 10: 112–119

Smith MC, Vetter JS, Alam SR (2005) Scientific computing beyond CPUs: FPGA implementations of common scientific kernels. In: MAPLD international conference

Sommer L, Weber L, Kumm M, Koch A (2020) Comparison of arithmetic number formats for inference in sum-product networks on FPGAs. In: IEEE symposium on field programmable custom computing machines (FCCM)

Srivastava N, Rong H, Barua P, Feng G, Cao H, Zhang Z, Albonesi D, Sarkar V, Chen W, Petersen P et al (2019) T2S-tensor: productively generating high-performance spatial hardware for dense tensor computations. In: IEEE symposium on field programmable custom computing machines (FCCM)

Stefanov T, Zissulescu C, Turjan A, Kienhuis B, Deprette E (2004) System design using Khan process networks: the Compaan/Laura approach. In: Design, automation, and test in Europe (DATE)

Stephenson M, Babb J, Amarasinghe S (2000) Bidwidth analysis with application to silicon compilation. In: ACM SIGPLAN notices

Stewart R, Duncan K, Michaelson G, Garcia P, Bhowmik D, Wallace A (2018) RIPL: a parallel image processing language for FPGAs. In: ACM transactions on reconfigurable technology and systems (TRETS)

Sujeeth AK, Lee H, Brown KJ, Rompf T, Chafi H, Wu M, Atreya AR, Odersky M, Olukotun K (2011) OptiML: an implicitly parallel domain-specific language for machine learning. In: International conference on machine learning (ICML)

Tan M, Dai S, Gupta U, Zhang Z (2015) Mapping-aware constrained scheduling for LUT-based FPGAs. In: International symposium on field-programmable gate arrays (FPGA)

Ustun E, Deng C, Pal D, Li Z, Zhang Z (2020) Accurate operation delay prediction for FPGA HLS using graph neural networks. In: International conference on computer-aided design (ICCAD)

Versal ACAP DSP Engine Architecture Manual (2020). https://www.xilinx.com/support/documentation/architecture-manuals/am004-versal-dsp-engine.pdf

Wang S, Kanwar P (2019) BFloat16: the secret to high performance on cloud TPUs. In: Google cloud blog

Wei X, Yu CH, Zhang P, Chen Y, Wang Y, Hu H, Liang Y, Cong J (2017) Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In: Design automation conference (DAC)

Zhang Z, Liu B (2013) SDC-based modulo scheduling for pipeline synthesis. In: International conference on computer-aided design (ICCAD)

Zhang C, Sun G, Fang Z, Zhou P, Pan P, Cong J (2018a) Caffeine: toward uniformed representation and acceleration for deep convolutional neural networks. In: IEEE transactions on computer-aided design of integrated circuits and systems (TCAD)

Zhang X, Wang J, Zhu C, Lin Y, Xiong J, Hwu W-M, Chen D (2018b) Dnnbuilder: an automated tool for building high-performance DNN hardware accelerators for FPGAs. In: International conference on computer-aided design (ICCAD)

Zhang J, Zhang W, Luo G, Wei X, Liang Y, Cong J (2019) Frequency improvement of systolic array-based CNNs on FPGAs. In: International symposium on circuits and systems (ISCAS)

Zhang Y, Pan J, Liu X, Chen H, Chen D, Zhang Z (2021) FracBNN: accurate and FPGA-efficient binary neural networks with fractional activations. In: International symposium on field-programmable gate arrays (FPGA)

Zhao R, Tan M, Dai S, Zhang Z (2015) Area-efficient pipelining for FPGA-targeted high-level synthesis. In: Design automation conference (DAC)

Zhao R, Song W, Zhang W, Xing T, Lin J-H, Srivastava M, Gupta R, Zhang Z (2017) Accelerating binarized convolutional neural networks with software-programmable FPGAs. In: International symposium on field-programmable gate arrays (FPGA)

Zhou Y, Al-Hawaj KM, Zhang Z (2017) A new approach to automatic memory banking using trace-based address mining. In: International symposium on field-programmable gate arrays (FPGA)

Approximate Computing Architectures

29

Muhammad Abdullah Hanif, Vojtech Mrazek,
and Muhammad Shafique

Contents

Approximate Computing 1028
Approximate Arithmetic Components 1030
Design Methodologies for Approximate Components 1030
Error Metrics and Evaluation Analysis for Approximate Components 1034
Design Methods for Building Approximate Hardware Accelerators: Case Studies for Error-Tolerant Applications 1037
Image and Video Processing Applications 1038
Deep Neural Networks (DNNs) 1046
Cross-Layer Approximations for Error-Tolerant Applications 1052
Methodology for Combining Hardware- and Software-Level Approximations 1052
Cross-Layer Methodology for Optimizing DNNs 1054
Case Studies for Improving the Energy and Performance Efficiency of DNN Inference . . . 1055
Structured Pruning 1055
Quantization 1057
Hardware-Level Approximations: Impact of Self-Healing and Nonself-Healing Designs on DNN Accuracy 1058
Conclusions 1063
References 1064

Abstract

Approximate computing is an emerging computing paradigm for improving the efficiency of error-tolerant applications. It allows designers to trade a negligible amount of accuracy for significant efficiency gains. This chapter provides an

M. A. Hanif (✉) · M. Shafique
Engineering Division, New York University Abu Dhabi, Abu Dhabi, United Arab Emirates
e-mail: mh6117@nyu.edu; muhammad.hanif@tuwien.ac.at; muhammad.shafique@nyu.edu

V. Mrazek (✉)
Faculty of Information Technology, Brno University of Technology, Brno, Czech Republic
e-mail: mrazek@fit.vutbr.cz

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_27

overview of approximate computing and how it can be exploited to offer improved efficiency while satisfying the user-defined accuracy/quality constraints. First, an overview of techniques for approximating arithmetic hardware modules is presented. Then, methodologies for efficient design space exploration of approximate modules and for building approximate accelerators are covered. Apart from hardware-level approximations, the chapter also discusses different software-level approximations and how they can be integrated with hardware approximations in a cross-layer design flow for building efficient systems.

Keywords

Approximate · Computing · Architecture · Deep neural network · DNN · Machine learning · Design method · Accelerator · Error-tolerant design · Image processing · Classification · Approximate circuits · Cross-layer

Approximate Computing

Emerging applications in the fields of cyber-physical systems (CPS) and Internet of things (IoT) have brought various challenges for the system design community. The major challenges include increases in the computational and memory requirements of applications, the growth in the number of computing devices, and the emergence of use cases with stringent energy/power constraints. All these challenges push for designing highly energy-efficient systems. Conventional techniques such as power gating and dynamic voltage and frequency scaling (DVFS) help in improving the energy efficiency of systems. However, they are insufficient to meet the growing demands of modern computing systems.

Studies by renowned research groups in the domain of energy-efficient computing systems have shown that a number of modern applications fall into the category of recognition, mining, and synthesis (RMS) applications that are (to some extent) resilient to errors (Nair 2014; Mishra et al. 2014; Esmaeilzadeh et al. 2012; Chippa et al. 2013). This error resilience is usually associated with one or more of the following factors:

1. Noise in real-world data
2. Inherent error masking characteristics of applications
3. Perceptual limitations of the users, e.g., a slight variation in the quality of an image/video (or audio) is usually unnoticeable by humans due to their psycho-visual/psychoacoustic limits
4. Absence of a unique nontrivial solution, e.g., web searches that result in slightly different but relevant links are usually equally acceptable

These factors can be exploited to relax the accuracy bounds of applications in order to achieve significant efficiency gains at the cost of minor accuracy/output quality loss.

Approximate computing (Xu et al. 2016; Shafique et al. 2016) is a computing paradigm that offers the opportunity to trade accuracy for improving the performance/efficiency of a system. This is mainly possible due to the extended design space that enables the designer to select designs having better efficiency compared to conventional designs while still meeting the user-defined accuracy constraints. Typical applications of approximate computing are in the areas of audio-visual data processing and machine learning, as slight variations in the output of these applications can be tolerated due to the intrinsic characteristics of the applications or perceptual limitations of the users.

Approximations can be applied at different layers of the HW/SW computing stack. At the software level, techniques like loop perforation (Sidiroglou-Douskos et al. 2011) and code simplification (Mohapatra et al. 2011) are commonly used, while, at hardware level, techniques like circuit approximation through voltage/frequency scaling (Chang et al. 2011) and functional approximations (Gupta et al. 2011) are widely used. Voltage/frequency scaling techniques can induce timing errors in the system, as in such cases the circuit operates at a lower voltage than the nominal value (Srinivasan et al. 2016). In functional approximation, the original circuit is replaced with a less complex substitute that exhibits almost the same functionality but improves nonfunctional circuit parameters such as power/energy consumption, latency, and area (Gupta et al. 2011).

This chapter provides an overview of approximate computing and how it can improve efficiency while satisfying the user-defined accuracy constraints. Figure 1 presents the overview of the chapter. First, section "Approximate Arithmetic Components" covers techniques for designing approximate arithmetic mod-

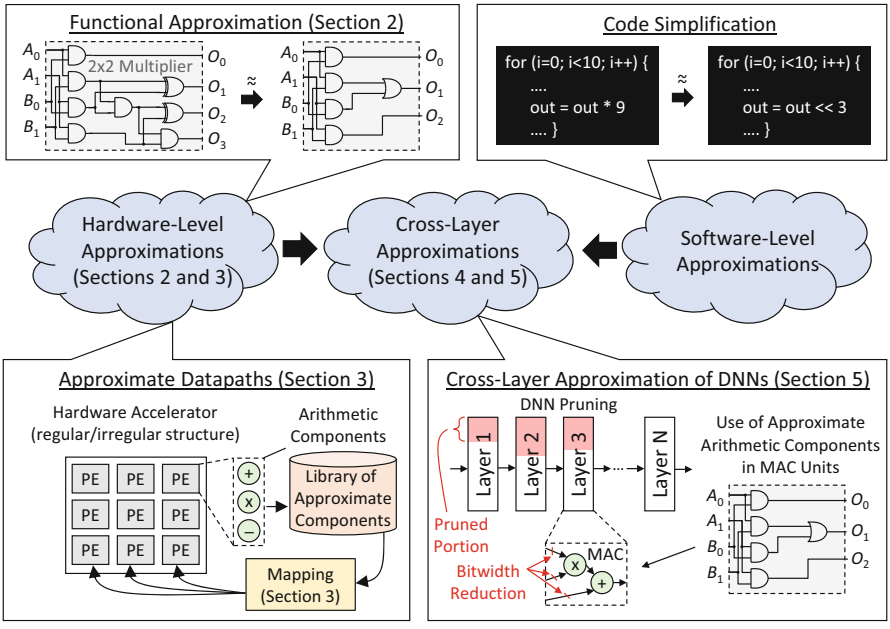


Fig. 1 Overview of the chapter

ules (e.g., adders and multipliers). Then section "Design Methods for Building Approximate Hardware Accelerators: Case Studies for Error-Tolerant Applications" presents design methodologies for automatically generating approximate datapaths for application-specific systems. Section "Cross-Layer Approximations for Error–Tolerant Applications" presents a cross-layer design flow that integrates software-level and hardware-level approximations. Toward the end, section "Case Studies for Improving the Energy and Performance Efficiency of DNN Inference" highlights the effectiveness of cross-layer approximations for deep learning applications, and section "Conclusions" concludes the chapter.

Approximate Arithmetic Components

The use of approximate computing techniques introduces an error, called *approximation error*, which is a measure of the difference between the exact computing solution and the approximate one. This error should allow the designers to reduce the power consumption of the circuits. The approximation error can be introduced on various levels. This section covers various techniques for designing approximate arithmetic components such as adders or multipliers. These components may be used as basic building blocks in high-level approximation methods introduced in the following sections.

Design Methodologies for Approximate Components

This work focuses on approximate arithmetic circuits because they are frequently used in the key applications relevant for approximate computing. The methods for functional approximations can be divided into two categories: (1) manual and (2) automated.

The *manual (ad hoc) methods* are developed for a specific circuit component. In this chapter, examples of manual approximation of two key arithmetic circuits – adders and multipliers – are described. These circuits are widely approximated because they realize key operations in applications requiring low-power processing. MACs, although widely employed, are typically approximated by using separate multiplier and adder units instead of introducing an error to the complex MAC circuit, and thus these circuits are not discussed in the chapter. Designers of manually approximated circuits found some regularities in the design and modified the structure or the truth table of the circuit (Fig. 2a). On the other hand, *automated methods* use general-purpose circuit resynthesis and approximation techniques and enable approximation of arbitrary circuits. These methods start with an original (exact) circuit and, typically iteratively, modify its structure as shown in Fig. 2.

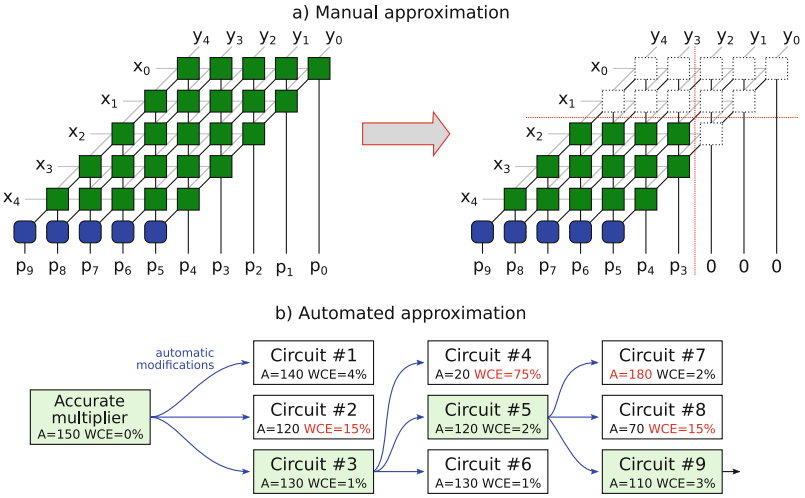


Fig. 2 Examples of two possible approaches for approximation of multiplier: (**a**) manual, where a designer found the rules for effectively omitting cells (Mahdiani et al. 2010), and (**b**) automated iterative approximation of a multiplier having the best area (A) and *worst-case error* (WCE) below 5%

Manual Approximation Methods

Adders: An adder performs the addition of two binary numbers. Two basic implementations are (1) ripple-carry adder (RCA), where the carry of each full adder is propagated to the next full adder, and (2) carry-lookahead adder (CLA), where several units working in parallel generate three signals ("sum," "propagate," and "generate") that are employed to quickly generate the carry-in signals. The CLA has significantly shorter delay than RCA. However, the area and power dissipation of CLA is larger than RCA. Many approximation principles for the adders implemented using one of these two schemes have been proposed in the literature (Jiang et al. 2017). The approximations can be classified into the following classes:

- *Speculative adders* were proposed by Lu (2004). In this architecture, the CLA structure is approximated using prediction of the carry for each sum bit.
- *Segmented adders*, where the addition is divided into n smaller subadders operating in parallel. These subadders have a fixed carry, and their delay is n-times shorter (Mohapatra et al. 2011). An advanced version divides the addition to the carry generation and sum generation, where each summation utilizes the information from the previous carry generation (Zhu et al. 2009).

- *Approximate carry select adders* consist of several subadders. Each subadder is made of two speculative adders – one with carry-in "0" and another with carry-in "1". The carryout of the first adder is connected to a multiplexor in the next block selecting the output of one of two speculative adders (Du et al. 2012).
- *Approximate full adders* are implemented in LSBs of the adder. For example, the simple use of OR gates instead of full adders and ignoring carries in the LSB part can lead to enormous power and time savings (Mahdiani et al. 2010).

Multipliers: Compared to addition, multiplication is a more complex operation. Generally, it consists of stages of partial product generation, accumulation, and final addition. There are several accurate multiplier architectures. The manually approximated n-bit multipliers are usually derived from one of the following schemes: (1) an array multiplier, where the sum and carry signals are generated by n-bit adders in each of n rows and they are passed to the adders in the next row, and (2) Wallace (or Dadda) tree multipliers dividing the multiplication into layers, where the adders work in parallel without any carry propagation within the layer. The array multiplier is smaller than the tree multiplier but slower. The approximations can be implemented in the following parts of the multipliers (Jiang et al. 2017):

- *Approximation in generating partial products* modifies the submultipliers, which the multiplier is composed of. For example, Kulkarni et al. proposed an approximate 2 × 2-bit multiplier where only one single entry is altered ($3 \times 3 = 7$ and the remaining ones are correct) (Kulkarni et al. 2011). Larger multipliers are designed using this 2-bit multiplier as a building block.
- *Approximation in the partial product tree* modifies the structure of the multiplies. This approach is utilized in the broken-array multiplier (Mahdiani et al. 2010). This multiplier omits some rows and columns in the array multiplier. A straightforward truncation of LSBs in operands (e.g., the usage of accurate 6-bit multiplier instead of the 8-bit one) also modifies the partial product tree by omitting some partial product cells. The omitting approach can be done in an adaptive way. In the multiplier proposed by Kyaw et al. (2010), the LSB cell function is controlled by the MSBs of operands.
- *Approximation in counters or compressors* in the partial product tree utilizes the tree structure of the multiplier. The key operations in each level are compressions, where 3 bits or 4 bits are compressed to 2 bits (3:2 or 4:2 compressors). These circuits can be approximated, for example, by a substitution of full adders by approximate ones (Momeni et al. 2015).

Automated Approximation Methods

SALSA: The Systematic methodology for Automatic Logic Synthesis of Approximate circuits (SALSA) is an automated approach that turns the approximation synthesis to the standard synthesis task (Venkataramani et al. 2012). A virtual circuit containing an accurate solution, a candidate circuit, and decision circuit (with one output) is constructed. The output is active when the error bound constraint is

violated. The *don't care states* are iteratively applied to the approximate solution. These states are accepted if the output of the virtual circuit remains zero for all input combinations. Thereafter, a traditional don't care-based optimization technique is applied.

SASIMI: Another systematic approach, Substitute-And-SIMplIfy (SASIMI) (Venkataramani et al. 2013), tries to identify signal pairs in the circuit that show the same value with high probability and substitutes one for the other. These substitutions result in some logic to be eliminated from the circuit. In addition to that, the downsizing of gates on critical paths (simplification) may be enabled. Moreover, the connection of the signal pairs using a configurable substitution circuit provides a kind of quality configurable circuit that can dynamically operate at different accuracy levels depending on the application requirements.

ABACUS: In contrast with previous automated methods, Automated Behavioral Approximate CircUit Synthesis operates on the HDL level. It automatically generates approximate circuits directly from the behavioral-level description. In order to perform desired approximations, the method modifies the abstract synthesis tree (AST) using the following operators: (1) simplification of data types, (2) substitution of arithmetic operations by approximate operations, (3) transformation of arithmetic expressions, (4) substitution of variables with constants, and (5) loop transformations. In each iteration of the algorithm, the operations are randomly applied to the accurate circuits, while the error bound is checked after the application (Nepal et al. 2014). The search algorithm is based on a simple hill-climbing algorithm or multi-objective NSGA-II algorithm (Nepal et al. 2017).

AIG-Rewriting: Another automatic synthesis approach uses And-Inverter Graph (AIG)-based rewriting. The AIG is a widely employed representation in logic synthesis. The algorithm identifies the longest paths in the circuit. Then *cuts* are selected by performing cut enumeration on the selected paths. In a logic circuit represented by an acyclic graph, a cut of node n is a set of nodes of the network, called leaves, such that each path from primary inputs to n passes through at least one leaf (Mishchenko et al. 2006). Each cut is replaced by an approximate cut (typically by zero constant) to generate a new candidate circuit. If the candidate meets the error constraints, it is accepted to the next iteration (Chandrasekharan et al. 2016).

ASLAN: Automatic methodology for Sequential Logic ApproximatioN (ASLAN) performs synthesis of approximate sequential circuits. The algorithm tries to identify combinational blocks in a sequential circuit that are amenable to approximations. Then, existing combinational approximation techniques are utilized to obtain a series of approximate versions having different quality levels. A gradient/descent approach is used to iteratively approximate the entire sequential circuit, while the overall error bound is checked using a formal verification approach (Ranjan et al. 2014).

BLASYS: Another methodology for approximate circuit synthesis based on Boolean matrix factorization (BMF) is BLASYS (BMF-based Logic Approximate SYnthesiS). A heuristic algorithm cuts the original circuits to small subcircuits. The truth table of a subcircuit of the design is approximated using BMF to a controllable approximation degree. The results of the factorization are used to synthesize a less complex subcircuit. A subcircuit design-space exploration technique helps to identify the best order for subcircuit approximations. The first version of this methodology (Hashemi et al. 2018) targeted Hamming distance only. However, in the most recent version (Ma et al. 2019), different error metrics are available. This tool is available as an open source at https://github.com/scale-lab/BLASYS.

Evolutionary Algorithm-Based Methods: The logic synthesis is based on small iterative changes of the initial circuit and optimizing the so-called fitness value. Vasicek and Sekanina successfully employed this idea for approximate circuit design by introducing the error metric to the fitness function (Vasicek and Sekanina 2015).

The main advantage of evolutionary approximation is that the heuristic searching algorithm can easily handle arbitrary constraints by giving penalties to the fitness function. Some penalties can be introduced for exceeding error metric (e.g., worst-case arithmetic error, mean relative error [MRE], etc.). Moreover, the evolutionary approximation can handle any application-specific constraint like accurate multiplication by zero (Mrazek et al. 2016) or nonuniform input distribution (Vasicek et al. 2019) as well.

The evolutionary approximation was also used in the context of FPGAs. GRATER tool (Lotfi et al. 2016) employs a genetic algorithm to determine the precision of variables within an OpenCL kernel. By selectively reducing the precision, the number of parallel approximate kernels that can be mapped in the fixed area budget of an FPGA can be increased with respect to the original kernel implementations.

Error Metrics and Evaluation Analysis for Approximate Components

The quality of approximate combinational circuits is typically expressed using one or several error metrics. In addition to the error rate, the average-case as well as the worst-case situation can be analyzed. Among others, the mean absolute error (MAE) and the MRE are the most useful metrics that are based on the average-case analysis. Selection of the right metrics is a key step of the whole design. When an arithmetic circuit is approximated, for example, it is necessary to base the error quantification on an arithmetic error metric. For general logic circuits, where no additional knowledge is available and where there is not a well-accepted error model, Hamming distance or error rate is typically employed.

The following paragraphs summarize the error metrics that have been employed in literature to quantify the deviation between the outputs produced by a functionally

correct design and an approximate design. These metrics are divided into two categories. The category of arithmetic errors consists of metrics that compare integer values of the circuit outputs. The Boolean error metrics are classified as general errors.

Arithmetic Error Metrics

Let $f : \mathbb{B}^n \rightarrow \mathbb{B}^m$ be an n-input m-output Boolean function that describes the correct functionality (the accurate function) and $f' : \mathbb{B}^n \rightarrow \mathbb{B}^m$ be an approximation of it, both implemented by two circuits, namely, F and F'.

The *worst-case arithmetic error*, sometimes denoted as *error magnitude* or *error significance* (Chan et al. 2013), is defined as

$$e_{wce}(f, f') = \max_{\forall x \in \mathbb{B}^n} |int(f(x)) - int(f'(x))|, \tag{1}$$

where $int(x)$ represents a function $int : \mathbb{B}^m \rightarrow \mathbb{Z}$ returning an integer value of the m-bit binary vector x. Typically, a natural unsigned binary representation is considered, i.e., $int(x) = \sum_{i=1}^{m} 2^i \cdot x_i$. The worst-case error represents the fundamental metric that is useful to guarantee that the approximate output differs from the correct output by at most error bound e.

In the literature, the *relative worst-case error*

$$e_{wcre}(f, f') = \max_{\forall x \in \mathbb{B}^n} \frac{|int(f(x)) - int(f'(x))|}{int(f'(x))} \tag{2}$$

is frequently employed to constrain the approximate circuit to differ from the correct one by at most a certain margin. Note that a special care must be devoted to the cases for which the output value of the original circuit is equal to zero, i.e., the cases when the denominator approaches zero. This issue can be addressed by either omitting test cases when $int(f(x)) = 0$ or biasing the denominator by 1. The first approach is usually employed in the manual approximation methods where the zero results are accurate (Jiang et al. 2017).

The *average-case arithmetic error* (also known as *MAE*) is defined as the sum of absolute differences in magnitude between the original and approximate circuit, averaged over all inputs:

$$e_{mae}(f, f') = 2^{-n} \sum_{\forall x \in \mathbb{B}^n} |int(f(x)) - int(f'(x))|. \tag{3}$$

If the expression in the sum is replaced by the equation for relative error distance, the *mean relative error* is calculated:

$$e_{mre}(f, f') = 2^{-n} \sum_{\forall x \in \mathbb{B}^n} \frac{|int(f(x)) - int(f'(x))|}{int(f'(x))}. \tag{4}$$

Note that the values produced by absolute error metrics e_{mae} and e_{wce} can be very large. Hence, these values can be expressed as a part of the output range using division by $2^m - 1$, i.e., the maximal output value. For example, the worst-case arithmetic error of 64 for an 8-bit output circuit (e.g., 4-bit multiplier) is equal to 25% error.

General Error Metrics

In addition to the arithmetic error metrics, there are metrics that are not related to the magnitude of the output of the correct or approximate circuit. These errors are typically used in approximation of general combinational circuits, where the weight of the output bits is unknown. In these circuits such as coders, decoders, and widely used benchmark circuits (e.g., ISCAS-89, ITC-99, etc.), the output value is not an arithmetic number, and arithmetic errors cannot be calculated (Vasicek and Sekanina 2014, 2015). However, the *error probability* is widely employed in arithmetic circuits as well.

The *error rate*, referred to as the *error probability*, represents the basic measure that is defined as the ratio of input vectors for which the output value differs from the original one:

$$e_{prob}(f, f') = 2^{-n} \cdot \left|\{\forall x \in \mathbb{B}^n : f(x) \neq f'(x)\}\right| \tag{5}$$

In many cases, it is also worth to consider the Hamming distance between $f(x)$ and $f'(x)$. The *worst-case Hamming distance*, denoted also as *bit-flip error* (Chen et al. 2014), is defined as

$$e_{bf}(f, f') = \max_{\forall x \in \mathbb{B}^n} \sum_{i=1}^{m} (f(x) \oplus f'(x))_i \tag{6}$$

and gives the maximum number of output bits that simultaneously output a wrong value. The average number of changed output bits, denoted as the *average Hamming distance*, can be expressed as follows:

$$e_{mhd}(f, f') = 2^{-n} \sum_{\forall x \in \mathbb{B}^n} \sum_{i=0}^{m} (f(x) \oplus f'(x))_i . \tag{7}$$

Quality Evaluation

In the error-metric formulas, the enumeration of all possible input vectors is employed. For a larger number of inputs n, it is not feasible to enumerate $\mathbb{B}^n$. This issue can be solved by (a) enumerating a subset of $\mathbb{B}^n$ or (b) obtaining the exact value using a formal verification approach. The formal verification can be performed by exhaustive simulation (with maximal instruction level of SIMD parallelization) (Hrbacek and Sekanina 2014; Mrazek et al. 2018) or some formal verification technique. These techniques typically construct a virtual miter circuit (consisting of candidate circuit, golden solution, and comparison circuit). Reduced Ordered Binary Decision Diagrams (ROBDD) or SAT conjunctive normal form (CNF)

representation was employed in the area of approximate circuits. The ROBDDs can help the users to determine various error metrics (Hamming distance (Vasicek and Sekanina 2014) or mean or worst-case arithmetic error (Soeken et al. 2016; Vasicek et al. 2017)). However, SAT solving is more effective for complex circuits like multipliers. These solvers allow only to determine if the worst-case error is below some given threshold, but very complex circuits such as 32-bit multipliers or 128-bit adders can be approximated (Češka et al. 2017).

Design Methods for Building Approximate Hardware Accelerators: Case Studies for Error-Tolerant Applications

Approaches for creating approximate components were presented in section "Design Methodologies for Approximate Components". The components are typically organized in libraries (e.g., Shafique et al. 2015; Hanif et al. 2017; Mrazek et al. 2017). These libraries contain from tens to thousands of approximate implementations for each arithmetic operation (e.g., 8-bit multiplication, 16-bit addition, etc.); the user is provided with a broad set of implementation options to reach the best possible trade-off between QoR (quality of results) and energy (or other hardware parameters) at the accelerator level.

If the user wants to use the components in his application, they start with some accurate accelerator, where the accurate operations are replaced by corresponding approximate components. However, it is intractable to find an optimal combination of approximate circuits, even for an accelerator consisting of a few operations. Identifying the most suitable replacements of the arithmetic operations of the target accelerator with the approximate circuit is a complex task. In this chapter, two approaches to this task are presented. As it is a multi-objective optimization problem, there is no single optimal solution; rather, multiple ones typically exist.

The designers are primarily interested in approximate circuits belonging to the *Pareto frontier* that contains the so-called non-dominated solutions. Consider two objectives to be minimized, for example, the mean error and energy. Circuit C1 (Pareto) *dominates* another circuit C2 if (1) C1 is no worse than C2 in all objectives and (2) C1 is strictly better than C2 in at least one objective.

This problem resembles the binding step of high-level synthesis (HLS), whose objective is to (i) map elementary operations of the algorithm to specific instances of components that are available in the component library and (ii) optimize hardware parameters such as latency, area, and power consumption. In the context of approximate circuits, the principal difference and difficulty lie in the QoR evaluation at the accelerator level. Except for some particular cases (e.g., Mazahir et al. 2017a,b), it is in general unknown how the errors propagate if two or more approximate circuits are connected in a more complex circuit. A common approach is to estimate the resulting error using either analytic or statistical techniques, but it usually is a very unreliable approach as seen in Li et al. (2015). If the problem is simplified in such a way that the only approximation technique is truncation, then an optimal number of bits to be approximated can be determined (Sengupta et al. 2017).

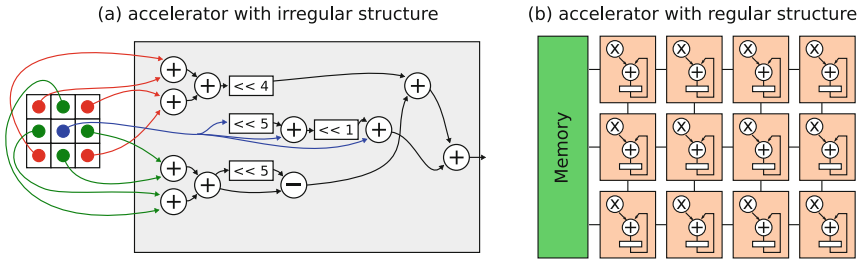


Fig. 3 Two types of accelerators: (**a**) with the irregular structure of fixed Gaussian filter having ten adders and one subtractor with different levels of approximation and (**b**) PE array for inference of neural network having PE employing the same adder and multiplier

Two major types of accelerators are discussed in the following sections. The first one maps every operation on every single hardware component (Fig. 3a). Typical examples of such irregular accelerators are image, video, or signal processing filter pipelines. The automated methodology shown in section "Image and Video Processing Applications" maps the approximate components to the operations. The second type of accelerator shares the hardware component for multiple operations (Fig. 3b). The sharing occurs, for example, in neural network inference acceleration. The layer operations are executed on a PE array where each processing element handles multiple different convolutions. In this case, additional constraints (e.g., only a few approximate PE arrays, order of the layers) must be satisfied. However, the structure of the neural network may be modified simultaneously. The approximation of neural networks is discussed in section "Deep Neural Networks (DNNs)".

Image and Video Processing Applications

Many different operations are employed in a typical image processing pipeline. In this section, three accelerators of different complexities that are typically used as benchmarks in image processing will be considered. In particular, Sobel edge detector (Sobel ED) (five operations), Gaussian filter with fixed coefficients (fixed GF) (11 operations), and generic Gaussian filter (generic GF) (17 operations) working on the 3×3 filter kernel were chosen.

Since there are hundreds to thousands of different approximate components for each operation and the complexity is exponential, the number is enormous. While the approximation of the five-operation accelerator is solvable by an exhaustive enumeration of all possible configurations, the accelerator consisting of 17 operations represents a nontrivial problem.

AutoAx Methodology

To address the approximate component binding problem, the authors proposed the AutoAx methodology (Mrazek et al. 2019a) that enables fast QoR and hardware cost

evaluation by means of machine learning algorithms and heuristic multi-objective searching algorithm.

The methodology requires the following inputs from the user: a hardware description of the chosen accelerator, corresponding software model, and training (benchmark) data. Hierarchical hardware as well as software models are expected in order to be able to replace relevant operations with their approximate versions and to evaluate how this change affects the QoR. Approximate circuits are taken from a library, in which each of them is fully characterized and many approximate implementations exist for each operation.

Let the accelerator contain n operations that can be implemented using some approximate circuits for the library. A *configuration* is referred to as a particular assignment of approximate circuits from the library to n operations of the accelerator. The goal of the methodology is to find a Pareto set of *configurations* where the design objectives to be optimized are QoR (e.g., SSIM, PSNR, etc.) and hardware cost (e.g., area, delay, power, or energy).

The whole process consists of three steps as illustrated in Fig. 4.

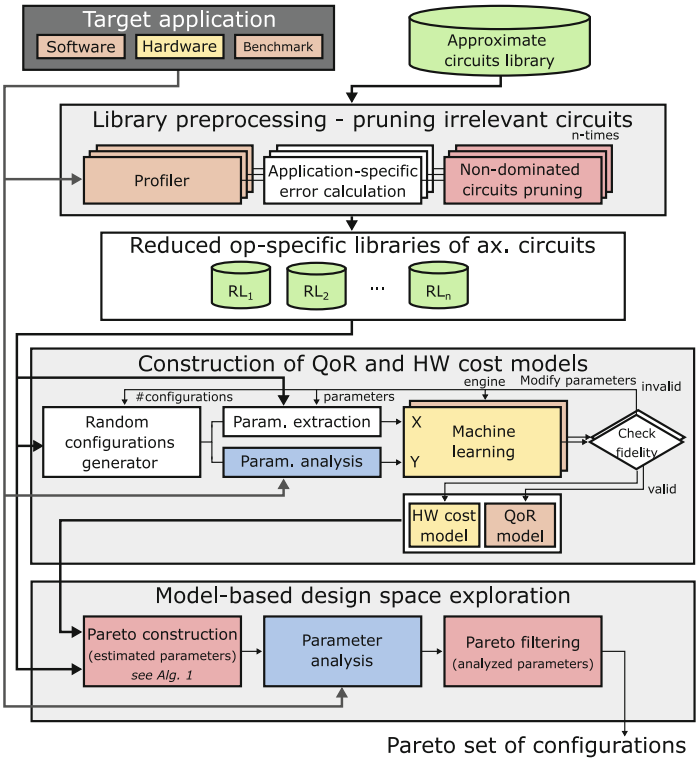


Fig. 4 Overview of the proposed *autoAx* methodology

Step 1: The library of the approximate circuits is preprocessed in such a way that clearly irrelevant circuits are removed. Irrelevant circuits are identified on the basis of their quality (measured with respect to a particular application) and hardware cost.

Step 2: Computational models enabling to estimate QoR and hardware cost are constructed by means of some machine learning algorithm. A small (randomly selected) subset of possible configurations is used for learning of the computational models.

Step 3: The Pareto frontier reflecting QoR and HW cost is constructed. To quickly remove as many low-quality solutions as possible, the construction algorithm employs the values estimated by the proposed models. The final Pareto front is then constructed using precisely computed QoR and hardware parameters by means of simulation and synthesis.

Library Preprocessing For each operation of the accelerator, a suitable subset of approximate circuits is separately identified in the library by means of benchmark data. For example, if the kth operation of the accelerator is 8-bit addition, then the objective of this step is to identify approximate 8-bit adders that form the Pareto front with respect to a suitable error metric (score) and hardware cost. The authors propose to base the selection on probability mass function (PMF) of the given operation which can be easily determined by simulation of the accelerator on benchmark data.

This process can be formalized as follows. Let I denote a set of all possible combination of values from the benchmark dataset that can occur on the input of kth operation $M(x_1, x_2, \dots)$, $x \in I$, $k = 1 \dots n$. Then, $D_k : I \rightarrow \mathbb{R}$ denoting the PMF of this operation is defined as $D_k(i_1, i_2, \dots) = Pr(x_1 = i_1 \wedge x_2 = i_2 \wedge \dots)$. This function is used to determine a score (weighted mean error distance) of an approximate circuit $\widetilde{M}$ implementing kth operation as follows: $\mathrm{WMED_k}(\widetilde{\mathrm{M}}) = \sum_{\forall i \in I} D_k(i) \cdot |M(i) - \widetilde{M}(i)|$. For each operation of the accelerator, this score is then used together with hardware cost to identify only those approximate circuits (i.e., 8-bit adders in our example) that are lying on a Pareto frontier.

Model Construction Since the synthesis and simulation are typically very time-consuming processes, it is intractable to use them to perform the analysis of hardware cost and QoR for every possible configuration of the accelerator. To address this issue, construction of two independent computational models is proposed – one for estimating QoR and a second for estimating hardware parameters. The estimation is based on the parameters of approximate circuits belonging to one selected configuration.

The models are constructed independently using a suitable supervised *machine learning algorithm* (regression problem). The learning process is based on providing example input–output pairs. In our case, each input–output pair corresponds with a particular configuration as shown in Fig. 5. One input is represented by a vector X,

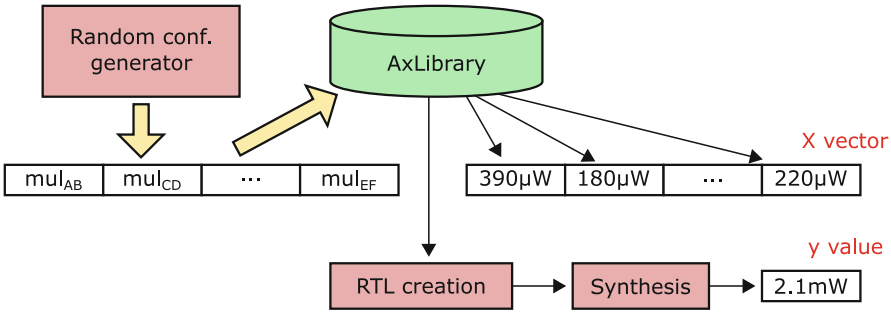


Fig. 5 Construction of training/testing set for ML model of hardware cost. The X-vector is extracted from the library (e.g., power, PDP [power–delay product] for HW cost, e_{mae}, e_{wce} for QoR), and the y-value is calculated using synthesis chain

which contains a subset of hardware or quality parameters of each approximate circuit realizing one of the operations as defined by the configuration. The output is a single scalar value y of QoR or hardware cost that is obtained by simulation and synthesis of the concrete accelerator with the given configuration. A training set typically containing from hundreds to thousands of configurations is generated for learning.

The goal of this step is to obtain high-quality models. A set of configurations different from the training set is used to determine the quality of the model and avoid *overfitting*, when the estimated values correspond too closely or exactly to training output values, and the model may, therefore, fail in fitting additional data. Typically, the accuracy is optimized by the machine learning algorithms. However, as the models are used for determining a relation between two different configurations, it is not necessary to focus on the accuracy. *Fidelity* (aka *monotonicity* (Bailey et al. 2007)) is considered as the optimization criterion that maximizes the fidelity of the model. The fidelity describes how often the estimated values are in the same relation ($<$, $=$ or $>$) as the real values for each pair of configurations. If the fidelity of the constructed model is insufficient, the parameters of the chosen learning algorithm should be tuned, or a different learning engine should be selected.

Model-Based Design Space Exploration In this step, the Pareto frontier containing those configurations that show the best trade-offs between QoR and hardware cost is constructed. In order to avoid time-consuming simulation and synthesis, the construction is divided into two stages. In the first stage, the computational models that were developed in the previous step are used to build a pseudo-Pareto set of potentially good configurations. In the second stage, based on the configurations forming the pseudo-Pareto set, a set of approximate accelerators is determined, fully synthesized, and analyzed by means of a simulator and benchmark data. A real QoR and real hardware cost is assigned to each configuration. Finally, these real values are used to construct the final Pareto set.

Algorithm 1 Pareto set construction

INPUT: RL – set of libraries, $RL = \{RL_1, RL_2, \cdots, RL_n\}$,
M_{HW} – HW costs model, M_{QoR} – quality model
OUTPUT: Pareto set $P \subseteq RL_1 \times RL_2 \times \cdots \times RL_n$

```
function HeuristicParetoConstruction(RL, M_QoR, M_C)
    Parent ← PickRandomlyFrom(RL_1 × RL_2 × ··· × RL_n)
    P ← ∅
    while ¬TerminationCondition do
        C ← GetNeighbour(Parent)
        e_QoR ← M_QoR(C)                          ▷ Estimate the quality of C
        e_HW ← M_HW(C)                            ▷ Estimate the HW costs of C
        if ParetoInsert(P, (e_QoR, e_HW), C) then
            Parent ← C
        else if StagnationDetected then           ▷ Parent not changed in last k iterations
            Parent ← PickRandomlyFrom(P)
        end if
    end while
    return P
end function
```

Although the first step reduced the number of possible configurations, the number of combinations may still be enormous especially for complex problems consisting of tens of operations. Therefore, the authors proposed an iterative heuristic algorithm (Algorithm 1) to construct the pseudo-Pareto set. The algorithm is a variant of stochastic hill climbing which starts with a random configuration (denoted as $Parent$), selects a neighbor at random (denoted as C), and decides whether to move to that neighbor or to examine another. The neighbor configuration is derived from $Parent$ by modifying a randomly chosen item of the configuration (i.e., another circuit is picked from the library for a randomly chosen operation). The quality and hardware cost parameters of C (e_{QoR} and e_{HW}) are estimated by means of appropriate estimation models. If the estimated values dominate those already present in Pareto set P, configuration C is inserted to the set, the set is updated (operation ParetoInsert), and the candidate is used as the $Parent$ in the next iteration. In order to avoid getting stuck in a local optimum, restarts are used. If the $Parent$ remains unchanged for k successive iterations, the $Parent$ is replaced by a randomly chosen configuration from P. The quality of the resulting Pareto set depends on the fidelity of the estimation models and on the number of allowed iterations. The higher fidelity, the better results. The number of iterations depends on the chosen termination condition. It can be determined by the size of P, execution time, or the maximum allowed number of iterations.

Results

The results are divided into two parts. Firstly, a detailed analysis of the results for the Sobel ED is provided to illustrate the principle of the proposed methodology. In the second part, only the final results are discussed due to the complexity of these problems and a limited space.

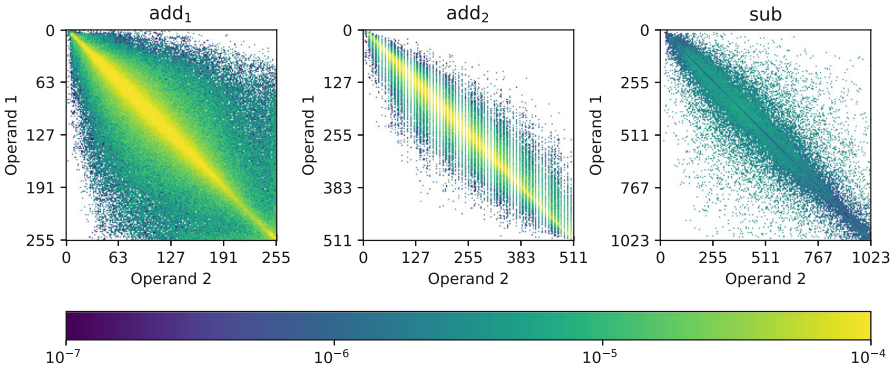


Fig. 6 PMF of operations in the Sobel ED

Sobel Edge Detector To eliminate irrelevant circuits from the library, a score is calculated for each circuit in the library. Firstly, the target accelerator is profiled with a profiler which calculates the PMF D_k for all operations (Fig. 6). Note that add_3 (resp. add_4) has almost identical PMF with add_1 (resp. add_2). Figure 6 shows that operand values (neighbor pixels) are typically very close. In the plot dealing with D_{add_2}, one can see regular white stripes caused by shifting of the second operand.

Using the obtained probabilities, the $WMED_k$ errors are calculated for all approximate circuits implementing kth operation. Then the components are filtered out. The process is guided by $area$ and $WMED_k$ parameters of the isolated circuits and keeps only Pareto-optimal implementations. At the end of this process, the number of circuits in reduced libraries is $|RL_{add_1}| = 35, |RL_{add_2}| = 32$, $|RL_{add_3}| = 37$, $|RL_{add_4}| = 33$, and $|RL_{sub}| = 36$.

The next step in the methodology is to construct models estimating SSIM and hardware parameters using parameters of the circuits belonging to one selected configuration. The $WMED$ of all employed circuits is employed as the input vector for the QoR model. For the hardware model, the input vector is $power, area$, and $delay$ of all circuits. Several learning engines are compared to identify the most suitable one for our methodology (1500 configurations for learning and 1500 configurations for testing were randomly generated using the reduced libraries).

The considered learning engines are the regression algorithms from *scikit-learn* tool for Python. Additionally, a naïve models are constructed for area ($M_a(C) = \sum_{\forall c \in C} area(c)$) and for SSIM ($M_{SSIM}(C) = -\sum_{\forall c \in C} WMED_k(c)$) to test if SSIM correlates with the cumulative arithmetic error and if the area correlates with the sum of areas of all employed circuits. These simple models are also considered in the comparisons.

Table 1 shows the fidelities for all constructed models when evaluated on the training and testing datasets. The best result for the testing datasets is provided by a random forest consisting of 100 different trees. The correlation between estimated and real area is shown in Fig. 7. The naïve models exhibit unsatisfactory results

Table 1 The fidelity of models for Sobel edge detector constructed by different learning engines

Learning algorithm	SSIM		Area	
	Train	*Test*	*Train*	*Test*
Random forest	99%	**96%**	97%	**92%**
Decision tree	**100%**	95%	**100%**	86%
K-neighbors	94%	94%	91%	89%
Bayesian ridge	90%	90%	91%	91%
Partial least squares	90%	90%	91%	90%
Lasso	90%	90%	91%	90%
Naïve model	–	90%	–	88%
AdaBoost	90%	90%	90%	88%
Least-angle	90%	90%	71%	72%
Gradient boosting	89%	89%	92%	91%
MLP neural network	86%	83%	92%	91%
Gaussian process	**100%**	71%	**100%**	55%
Kernel ridge	41%	42%	90%	90%
Stochastic gradient descent	24%	25%	75%	74%

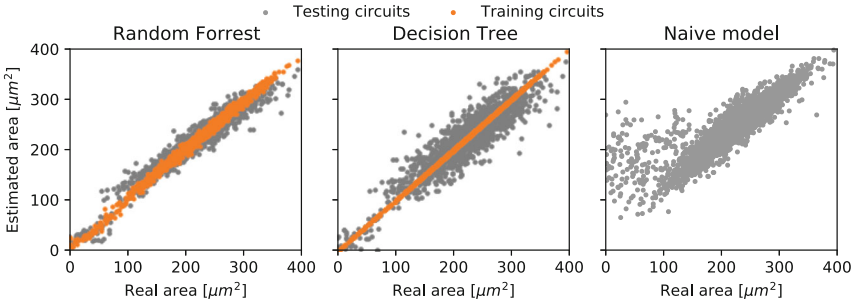


Fig. 7 Correlation of estimated area and real area obtained by synthesis tool for the selected learning engines used in Sobel ED experiment

especially for small resulting approximate accelerators. In the analysis of some of these cases in detail, the authors observe that the inaccuracy was typically caused by the last operation in the application (i.e., *sub*). As this operation shows a big error, it is significantly simplified by the synthesis tool, and as a consequence of that, many other circuits are removed from the circuit because their outputs are no longer connected to any component. Hence, the real area of these circuits was significantly smaller than the area calculated using the library. Due to this elimination, machine learning methods based on conditional structures (e.g., trees) exhibit better performance than methods primarily utilizing algebraic approaches (e.g., MLP NN).

The impact of input parameters on the model quality was analyzed. Including different error metrics such as the error variance did not improve the fidelity of QoR models. In contrast, omitting power and delay in hardware modeling led to 2% lower fidelities of these models on average.

The quality of the proposed heuristic algorithm that was used for Pareto frontier construction is evaluated now. Because of a low number of operations in Sobel ED, all possible configurations derivable from the reduced libraries RL_k (i.e., $4.92 \cdot 10^7$ configurations in total) can be evaluated. The proposed algorithm with a reasonable number of evaluations (10^5) could find the suboptimal solutions that are very close to the optimal ones. The proposed algorithm found solutions in three orders of magnitude closer to the optimal than the standard random search.

More Complex Pipelines The methodology was also applied to obtain approximate implementations of two versions of Gaussian image filter (fixed GF and generic GF). After profiling this accelerator and reducing the library of approximate circuits accordingly, random forest-based models of QoR and hardware parameters were created using 4000 training and 1000 testing randomly generated configurations. In the case of fixed GF, the fidelity of the area estimation model is 87% for hardware parameters and 92% for QoR. The fidelity of both models of generic GF is 89%. If the synthesis and simulations run in parallel, the detailed analysis of one configuration takes 10 s on average, and the model-based estimation of one configuration takes 0.01 s on average.

The Pareto construction algorithm evaluated 10^6 candidate solutions. On average, 39 iterations were undertaken to find a new candidate suitable for the Pareto front.

Table 2 shows the size of the design space after performing particular steps of the proposed methodology. For example, there are $7.15 \cdot 10^{63}$ configurations in the generic GF design space. The elimination of irrelevant circuits in the library reduced the number of configurations to $3.75 \cdot 10^{23}$. The number of configurations is enormous, and it would take 10^{17} years to analyze them. In contrast, the construction of 4000 random solutions for training of the models takes approximately 11 h, 10^6 iterations of the proposed Pareto construction algorithm employing the models takes 3 h, and the remaining 1000 configurations are analyzed in 3 h. Finally, approximately 100 configurations that are Pareto optimal in terms of area, SSIM, and energy are selected. In total, the proposed approach takes 17 h on a common desktop. Hypothetically, if the analysis would be used instead of the estimation model in the Pareto front construction, the analysis of 10^6 configurations would take 115 days.

Figure 8 compares resulting Pareto fronts obtained using the proposed methodology (orange line), the RS-based Pareto front construction algorithm (blue line), and the uniform selection approach (black line). The uniform selection approach

Table 2 Size of the design space after performing particular steps of the proposed methodology

	# configurations			
Application	*All possible*	*Lib. preprocessing*	*Pseudo-Pareto*	*Final Pareto*
Sobel ED	$1.96 \cdot 10^{15}$	$4.92 \cdot 10^7$	335	62
Fixed GF	$7.35 \cdot 10^{34}$	$1.73 \cdot 10^{16}$	1166	132
Generic GF	$7.15 \cdot 10^{63}$	$3.75 \cdot 10^{23}$	946	102

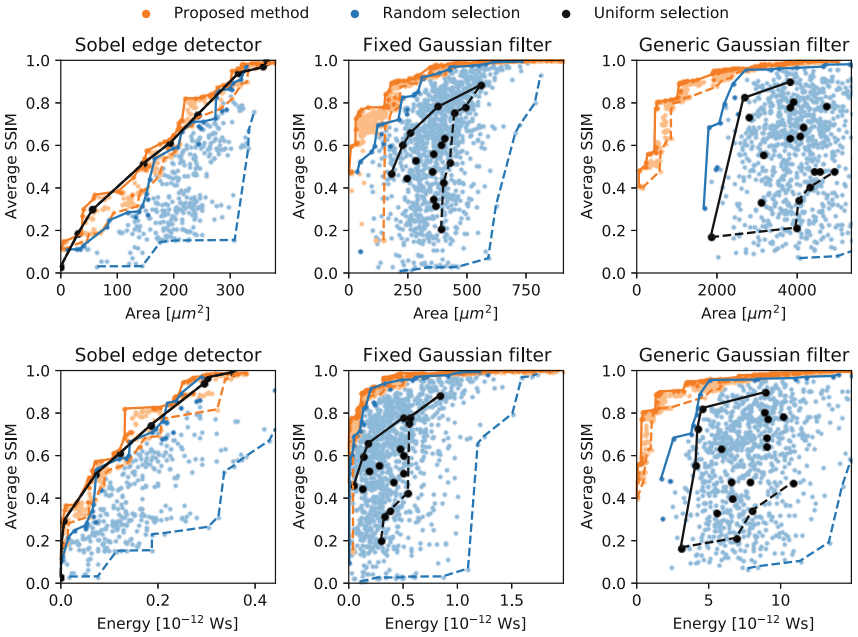


Fig. 8 Pareto fronts showing best trade-offs between SSIM, area, and energy obtained using three methods (orange, the proposed method; blue, random search; black, uniform selection) for three approximate accelerators

is a manual selection method which one would probably take if no automated design methodology is available. In this method, particular approximate circuits are deterministically selected to exhibit the same error WMED (relatively to the output range). Figure 8 shows that this method provides relevant results only for accelerators containing a few operations. The randomly generated configurations (blue points) were obtained from a 3-h run of the random configuration generation-and-evaluation procedure. They are included in these plots in order to emphasize high-quality solutions obtained by the proposed method.

Deep Neural Networks (DNNs)

The neural networks have come to be an important part not only of supercomputers but even small embedded systems realizing machine learning *on the edge*. The structure of hardware accelerators is different in contrast to the typical signal processing pipeline introduced in the previous section. The accelerator is organized as an array of processing elements. An arbitrary approximate component cannot be assigned to any layer of DNN because the number of the *tiles* (parts of the PE array) is limited. A significant proportion of energy is consumed by the computational path consisting primarily of multiplications (25–50% (Judd et al. 2018)).

The energy cost of the computational path can be reduced using approximate computing because the DNNs exhibit error resilience property. The standard approach is to assign the approximate components to the layers while considering PE array construction constraints. The promising alternative approach is to construct the architecture with approximate components (neural architecture search) (Pinos et al. 2021), but this approach is computationally intensive. Therefore, the authors proposed ALWANN methodology (Mrazek et al. 2019b) that assigns the approximate components with the help of a multi-objective evolutionary algorithm.

ALWANN Methodology

ALWANN requires the following inputs from the user: already trained NN being subject of the approximation, a library of basic approximate components (adders, multipliers), and knowledge of the architecture of the final HW accelerator. Two HW-based architectures (as discussed in the previous section) are considered in this work: *pipelined* and *power-gated* arrays. For simplicity, the MAC units will be implemented using accurate addition and approximate multiplication, but approximate addition can be introduced as well in general. Let $L = \{L_1, L_2, \ldots\}$ be a set of indexes of convolutional layers of NN and M be a set of available approximate w-bit multipliers. The user should specify the number of different tiles $|T|$ the accelerator will consist of. Typically, $|T| < |L|$ and $w = 8$ is sufficient. Each tile's NFU consists of the array of the same MAC units. Each layer L_i is supposed to be executed on a single tile T_j.

The method outputs a set of AxNNs (modified original NN together with the corresponding configuration of the HW accelerator tiles) that are Pareto optimal with respect to the energy consumption and classification accuracy. The approximations are introduced to the original NN by *replacement* of the accurate convolutional layers by approximate ones together with *weight tuning*. Considering the structure of the HW-based accelerator, two tasks are solved simultaneously. The methodology looks for the *assignment of the approximate multipliers* to MACs in SA tiles $T = \{T_1, T_2, \ldots\}$, i.e., mapping $map_{TM} : T \rightarrow M$, and for the *assignment of the convolutional layers* to SA tiles, i.e., mapping $map_{LT} : L \rightarrow T$. The weights in each layer are updated according to the properties of a particular multiplier assigned to the tile which computes the output of the layer.

The overall architecture of the proposed framework is shown in Fig. 9. The framework expects that a fully specified NN is available (typically in *protobuf* format). If not already done, the NN is firstly quantized to avoid floating point MAC operations. The protobuf specification of the quantized NN is then edited, and all convolutional layers are replaced by approximate ones. This step is necessary to have the ability to specify which multiplier should be used to calculate the output of the MACs separately for each layer. To obtain a Pareto set of various AxNNs, the authors propose to use multi-objective genetic algorithm (NSGA-II) (Deb et al. 2002). The algorithm maintains a population of $|P|$ candidate solutions represented as a pair (map_{TM}, map_{LT}). The search starts from an initial population which

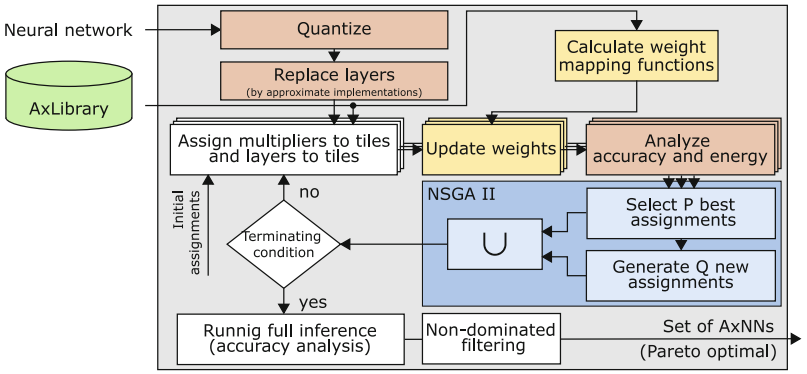


Fig. 9 Overall architecture of ALWANN framework

is generated either deterministically or randomly. The candidate solutions are iteratively optimized with respect to the accuracy of AxNN and energy required to perform one inference. For each candidate solution, a corresponding protobuf is created. This step includes the assignments of the multipliers to each approximate layer according to the map_{TM} and map_{LT} and refinements of the weights in each approximate layer depending on the chosen multiplier. Then, energy as well as quality of the obtained AxNN is evaluated on a subset of training data. The usage of the subset of training data reduces the testing time, and it simultaneously avoids overfitting. At the end of the optimization process when a terminating condition is met (typically the maximum number of allowed iterations is exceeded), the quality of the candidate solutions is evaluated using the complete training set. Solutions whose parameters are dominated by at least one other solution are filtered out.

In contrast to the AutoAx methodology for a generic pipeline, the introduced ALWANN approach does not employ ML models. The HW cost is estimated as a sum of energies because the chained approximation does not affect the overall HW cost in direct result sharing. Similarly, a fast evaluation of quality (classification accuracy) has been proposed. Since many approximate units work in parallel, this task can be performed on a GPU in a reasonable time. The common part of both methodologies is that they use a multi-objective genetic heuristic algorithm (a variant of NSGA-II).

Representation of Candidate AxNNs Each candidate solution is uniquely defined by a pair (map_{TM}, map_{LT}). The authors propose to use an integer-based encoding. The first part map_{TM} is encoded using $|T|$ integers where each integer corresponds with index i of multiplier $M_i \in M$. Similarly, the second part is encoded using $|L|$ integers where each integer determines index i of a tile $T_i \in T$ that will be used to compute the output of the corresponding layer. Depending on the structure of the chosen HW accelerator, additional restrictions may be applied. For example, for pipelined architecture, a rule that the tiles are assigned consequently can be constrained.

Evaluation and Experiments

To evaluate ALWANN, TensorFlow framework was extended to support approximate quantized layers. The extension has been published as open source at https://github.com/ehw-fit/tf-approximate. The tool flow is shown in Fig. 10. At the beginning, the common `QuantizedConv2D` layers are replaced with newly introduced `AxConv2D` layers. The remaining part follows the scheme already described in section "ALWANN Methodology". For the evaluation, ResNet networks (v1 with non-bottleneck blocks) (He et al. 2015) were chosen and trained to recognize images from CIFAR-10 dataset. The library of approximate multipliers consists of all 36 eight-bit fully characterized multipliers from the publicly available EvoApproxLib library (Mrazek et al. 2017).

Figure 11 shows the quality of AxNNs obtained using ALWANN from the original ResNet-8. The results are compared with three configurations of AxNNs mentioned in the previous section, especially to *uniform structures* widely used in the recent literature. The proposed method delivers significantly better AxNNs compared to the manually created AxNNs. The uniform structure (all layers approximated) widely used in the literature (see, e.g., Sarwar et al. 2018; Mrazek

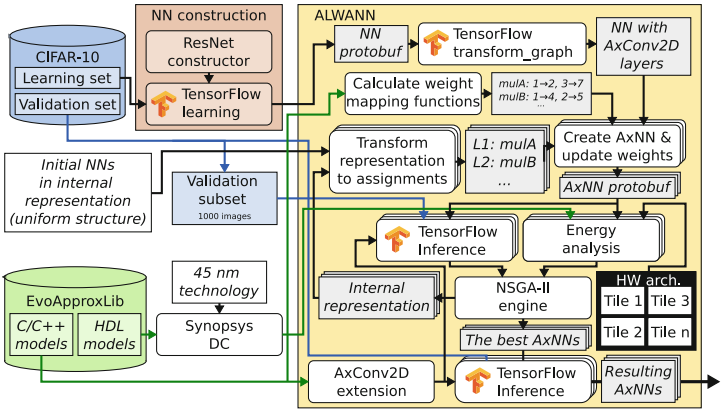


Fig. 10 Our tool flow for retraining-less approximation of ResNet neural network

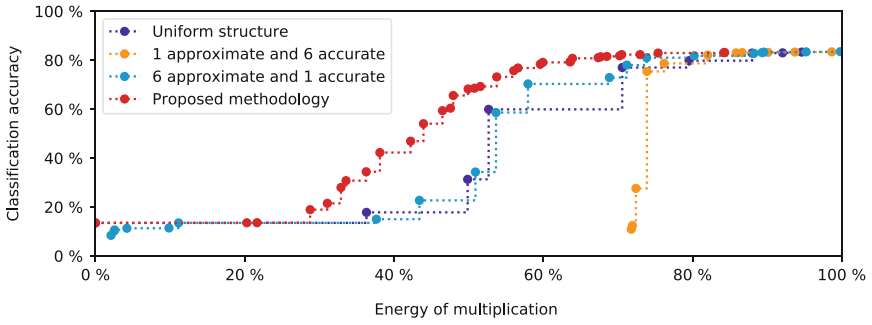


Fig. 11 Comparison of AxResNet-8 approximate neural networks constructed by means of proposed algorithm and NNs having a regular structure

et al. 2016) achieves results comparable to AxNNs with all but one approximated layers. In contrast to that, AxNN with one approximate layer leads to significantly worse results because of small energy saving. The proposed method provides **better trade-offs between the accuracy and energy consumption** in comparison with the uniform NN architectures reported in the state-of-the-art works.

A bottleneck of the algorithm was the expensive simulation of approximate multipliers on CPU. Although the multipliers were cached, our single core application has 10× lower performance than vectorized accurate multiplication. Since one inference of full dataset took 54.5 min, 7.5 days were needed for the construction of the approximate neural network. This problem was addressed in Vaverka et al. (2020) by employing approximate operations on a GPU. The speed was improved more than 200×, and the most complex 50-layer NN can be approximated in less than 2 h on a single GPU.

Overall Results Table 3 gives some parameters of the best AxNNs constructed using the proposed tool. The following parameters are reported for each network: relative accuracy and total and relative energy of convolutional operations. The relative values are calculated with respect to the original quantized (8-bit) ResNet. The quality of the obtained AxNNs for ResNet-50 is very promising. If a target application is able to tolerate 1% accuracy drop (from 89.15% to 88.1%), for example, more than 30% of energy can be saved. The evaluation across different architectures shows that it is not advantageous to use AxNNs having more than

Table 3 Parameters of selected AxNNs implementing dataset CIFAR-10. The relative values are compared to accurate 8-bit neural network, and total energy is related to the energy of one accurate multiplication E_M

AxNN	Accuracy	Relative accuracy	Relative energy	Total energy [$\times E_M$]
AxResNet-50	*89.15%*	*100.00%*	*100.00%*	*120.27 M*
	89.30%	100.17%	83.29%	100.17 M
	89.08%	99.92%	78.47%	94.37 M
	88.69%	99.48%	77.97%	93.77 M
	88.58%	99.36%	70.02%	84.21 M
	88.10%	98.82%	69.12%	83.13 M
	87.77%	98.45%	67.36%	81.02 M
	85.00%	95.34%	57.74%	69.45 M
AxResNet-14	*85.55%*	*100.00%*	*100.00%*	*35.33 M*
	85.87%	100.37%	80.32%	28.38 M
	85.42%	99.85%	74.34%	26.27 M
	84.77%	99.09%	70.85%	25.04 M
	83.82%	97.98%	64.64%	22.84 M
AxResNet-8	*83.26%*	*100.00%*	*100.00%*	*21.18 M*
	83.16%	99.88%	84.31%	17.86 M
	81.79%	98.23%	70.23%	14.87 M
	79.11%	95.02%	59.95%	12.70 M
	75.71%	90.93%	56.04%	11.87 M

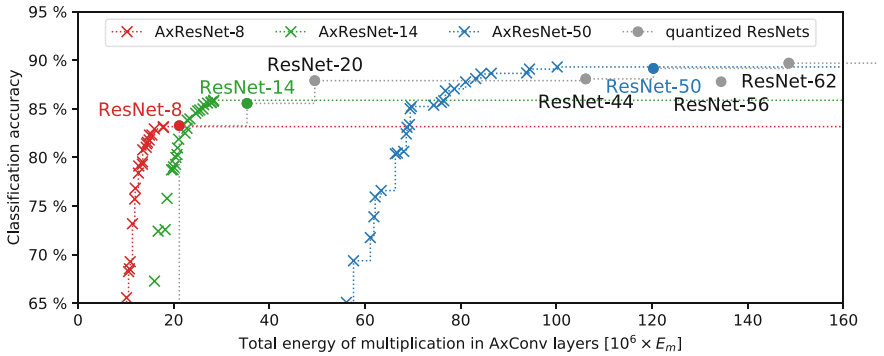


Fig. 12 Comparison of proposed AxNNs (crosses) with accurate quantized NNs (points) – the energy reports the energy of multiplications in the convolutional layers, while E_m is energy of one multiplication. Gray points represent quantized networks that were not approximated (complexity reduction)

4% (2% for AxResNet-14) degradation of accuracy for AxResNet-50, because AxResNet-14 (AxResNet-8) exhibit the same quality but lower energy.

Complete overview of the best obtained AxNNs having accuracy higher than 65% is provided in Fig. 12. In addition to the parameters of the AxNNs for three ResNet architectures discussed so far, the parameters of all possible ResNet architectures up to 62 layers (see the dots) are included, namely, ResNet-20, ResNet-44, ResNet-56, and ResNet-62, that have been trained in the same way as the remaining ResNet NNs. These NNs have been obtained by reducing the number of layers by multiples of six, i.e., at block boundaries. In total, seven different ResNet architectures are included. As evident, our method is able to produce significantly more design points; more than 40 points are produced from a single ResNet. Moreover, majority of the design points are unreachable by simple reduction of the number of layers (see the blue crosses vs. dot symbols). Considering the computational complexity, each ResNet instance must be trained separately. For complex structures, training of a new structure can take several days or weeks on computer clusters.

Comparison with State of the Art (SoA) Table 4 compares the proposed approach with the state-of-the-art approaches for reducing the energy of NNs that have been evaluated on CIFAR-10 dataset. Table 4 includes reported energy reduction and accuracy degradation. The requirement for retraining, uniformity of the architecture, and complexity of NN are also provided. In contrast with multiplier-less multiplication where only four different architectures were proposed (Sarwar et al. 2018), our approach allows to find a new design points with high granularity without retraining. Besides that, our approach enabled the authors to find AxNNs with low energy exhibiting low accuracy, e.g., <80%. Even these solutions can be beneficial, for example, as one of initial stages of some progressive chain classifier (Choi and Venkataramani 2019).

Table 4 Comparison of automated NN approximation methods: architectural parameters, energy and accuracy reduction reported on CIFAR-10

Approach	Retrain./Unif./Depth	Energy/Accuracy
Venkataramani (Venkataramani et al. 2014)	Yes/no/low	−22%/−0.5%
		−26%/−2.5%
Sarwar (Sarwar et al. 2018)	Yes/yes/high	−33%/−1.8%
He (He et al. 2015)	Yes/yes/high	$-12\%/-1.2\%_{\ 50\rightarrow 44}$
		$-71\%/-4.0\%_{\ 50\rightarrow 14}$
		$-48\%/-2.7\%_{\ 14\rightarrow 8}$
ALWANN (Mrazek et al. 2019b)	No/no/high	$-30\%/-0.6\%_{\ \text{AxRN-50}}$
		$-30\%/-0.9\%_{\ \text{AxRN-14}}$
		$-30\%/-1.7\%_{\ \text{AxRN-8}}$

Cross-Layer Approximations for Error-Tolerant Applications

Section covered effective techniques for hardware-level approximations. However, improvements can be achieved through software-level approximations as well. Therefore, this section presents a methodology for combining software- and hardware-level approximations to achieve significant improvements in the performance of a system by leveraging the error resilience characteristics of the application. After presenting a generic methodology for cross-layer approximation in section "Methodology for Combining Hardware- and Software-Level Approximations", section "Cross-Layer Methodology for Optimizing DNNs" covers a methodology specifically designed for optimizing DNN-based systems.

Methodology for Combining Hardware- and Software-Level Approximations

To design highly resource-efficient systems by exploiting the error resilience of the applications, it is necessary to employ approximations at both the software and the hardware levels. Figure 13 presents a cross-layer methodology for designing such systems where approximations are systematically employed across the hardware and the software stacks (Shafique et al. 2016). First, different approximation possibilities are explored at individual levels to short-list a set of Pareto-optimal points. This set can be a result of only a single type of approximation (e.g., functional approximation of computational modules) or multiple types (e.g., functional approximation of computational modules and voltage-scaling in on-chip memory) from the same level. In case of multiple types, efficient design space exploration methodologies are required to find the configurations that offer the best quality-efficiency trade-off. Once the points at individual levels are selected, they are forwarded for a cross-layer design space exploration to select a combination that offers the best efficiency while meeting the user-defined quality constraints. The

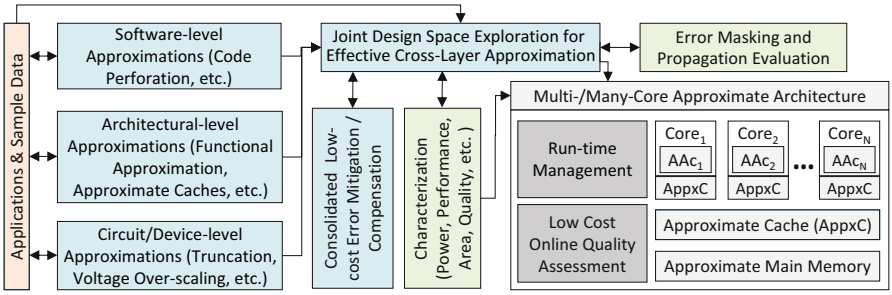


Fig. 13 A cross-layer approximation methodology for designing highly efficient systems (Shafique et al. 2016)

joint exploration is supported by fast error estimation methodologies that take into consideration error masking and propagation properties of approximations to estimate the joint effect of different approximations on the output quality. Low-cost error compensation modules can be employed to compensate for a portion of the quality loss. A set of points are then forwarded to the characterization stage for estimating the performance characteristics of the designs, e.g., power/energy and area. These characteristics are then used together with quality estimates to identify the optimal configurations that offer the best quality-efficiency trade-off. The system can also be equipped with an online approximation management module that can configure the system based on the user requirements and/or run-time conditions to maximize efficiency gains.

Note that most of the works in the domain of approximate computing are focused toward designing techniques for a specific layer of the computing stack, and only a limited amount of research has been carried out on cross-layer methodologies that systematically employ approximations at all the abstraction layers to achieve optimal quality-efficiency trade-off. This is mainly because there are several critical challenges in realizing an effective cross-layer methodology such as the one shown in Fig. 13. A few of these challenges are listed below:

- Designing methodologies/analytical models for evaluating the error masking and propagation characteristics of approximations, specifically for projecting them across layers of the computing stack.
- Designing techniques for efficiently estimating the overall performance characteristics of an approximated system that has different types of approximations deployed at different layers.
- Developing methods for low-cost consolidated error detection and correction for cross-layer approximations.
- Developing low-cost systems for online quality assessment and resource management. Such systems are mainly for applications that have high run-time variations and require modules for dynamically orchestrating the approximation knobs to achieve best efficiency while meeting the user-defined quality constraints.

Cross-Layer Methodology for Optimizing DNNs

DNNs are widely being used in many applications due to their state-of-the-art performance (LeCun et al. 2015). Studies have shown that they are (to some extent) resilient to errors in intermediate computations. This property of DNNs can be exploited through different types of approximations to reduce their execution cost and enable their deployment on resource-constrained devices. Toward this, various software-level and hardware-level approximation/optimization techniques have been proposed. At the software level, pruning and quantization are employed to reduce the complexity of the network and computations (respectively), and at the hardware level, customized hardware accelerators and approximate arithmetic modules are employed (as also shown in section "Deep Neural Networks (DNNs)"). These techniques can be combined in a systematic manner to achieve high efficiency gains. Figure 14 presents a cross-layer methodology that combines pruning and quantization techniques with hardware-level optimizations (Hanif and Shafique 2021). The methodology consists of the following steps:

- **Pruning:** At the software level, the most effective technique for optimizing DNNs is pruning. It involves removing the ineffectual weights from the network to reduce the complexity of DNNs. Based on its effectiveness, the cross-layer methodology employs pruning as Step 1. An iterative pruning technique is mainly employed that reduces the number of parameters in multiple iterations, where each iteration is (optionally) followed by partial retraining to compensate for the accuracy loss. The weights to be removed are selected based on their saliency, which can be estimated using L1-norm/L2-norm or by using a complex back-propagation algorithm. The number of weights removed in each iteration and the amount of retraining after each iteration are two key hyper-parameters that can impact the compression and/or accuracy of the resultant network and, therefore, have to be selected carefully. The iterations are performed till the accuracy of the network drops below the user-defined accuracy constraint, and

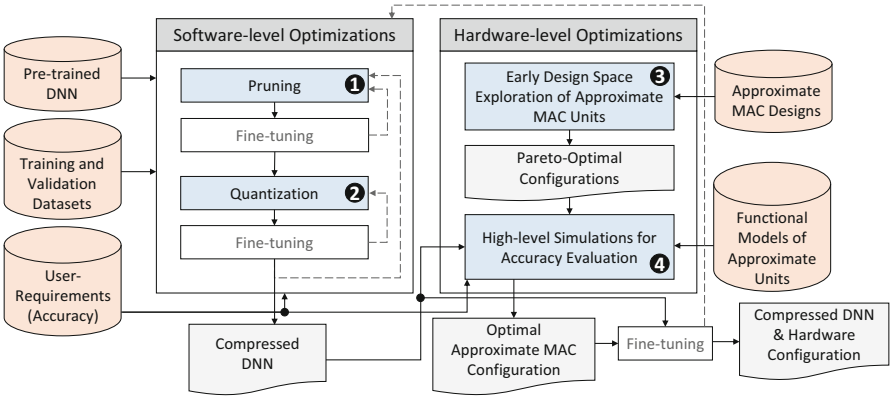


Fig. 14 A cross-layer optimization flow for DNNs (Hanif and Shafique 2021)

the network from the second to the last iteration is forwarded to the next step for further optimization.

- **Quantization:** The precision of DNN data structures impacts the memory requirements and the complexity of the computational modules. Quantization is employed to represent weights and activations using low-precision fixed-point format. It not only reduces the memory requirements for the inference stage but also helps in simplifying the hardware modules, e.g., MAC units. Therefore, the methodology employs quantization in Step 2 to further compress the network and simplify the logic units at the hardware level. The quantization process can be coupled with retraining to compensate for the accuracy loss due to quantization errors in the computations. Moreover, pruning and quantization can also be combined in a single unified process (Tung and Mori 2018). However, such methods require sophisticated optimization algorithms to efficiently explore the combined design space and propose an effective solution.
- **Hardware Approximations:** Specialized hardware accelerators are used for energy-efficient processing of data in real-world systems. These accelerators can be equipped with approximate units to further boost the efficiency gains. Toward this, Step 3 of the methodology explores the potential of hardware-level approximations, e.g., functional approximation of adders and multipliers. This step performs design space exploration of approximate modules to find the most suitable configurations that offer high efficiency while meeting the user-defined quality constraints. The step also explores the potential of internal self-healing modules, as they can offer better error characteristics in case of vector operations. These approximations can also be coupled with retraining to partially compensate for the accuracy loss due to approximations.

Case Studies for Improving the Energy and Performance Efficiency of DNN Inference

Structured Pruning

This section highlights the effectiveness of the pruning step (i.e., Step 1 in Fig. 14) for improving the efficiency of DNN inference. Figure 15 presents the flow considered in this study for pruning filters/neurons from a pre-trained DNN. The main steps of the flow are:

1. Given a pre-trained DNN, first, the methodology computes the saliency of each filter/neuron of the network using a suitable saliency measure, e.g., L1-norm.
2. Then for each layer of the DNN, it creates a copy of the network and removes $x\%$ of the least significant filters/neurons from the layer while keeping all rest of the layers intact.
3. The methodology then computes the accuracy and compression ratio of each model and registers them in θ. Note that for fast execution of the methodology, only a subset of the validation dataset is used to estimate the accuracy.

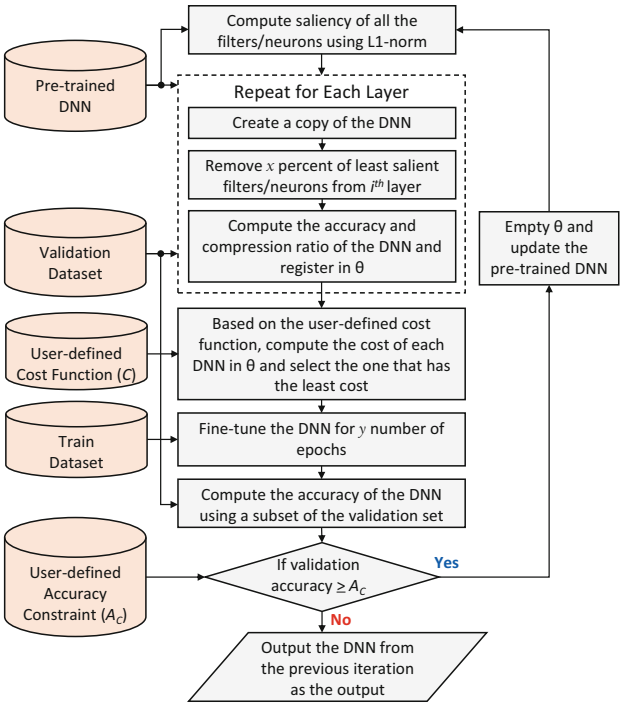


Fig. 15 The considered structured pruning methodology (Hanif and Shafique 2021)

4. A user-defined cost function C is then used to compute the cost of pruning in each individual layer.
5. The models in θ are then sorted based on their costs, and the one that has the least cost is selected, and all rest of the models are discarded.
6. The selected model is then fine-tuned for y number of epochs, and its accuracy is estimated using a subset of the validation dataset.
7. The accuracy is then compared with the user-defined accuracy constraint (A_c). If the accuracy is greater than the user-defined constraint, the pre-trained model is replaced with the pruned model, and the complete process is repeated until the accuracy falls below A_c. Once the accuracy is below A_c, the output of the previous iteration is passed as the final output of the methodology.

To show the effectiveness of pruning, the above flow is employed to prune filters/neurons from the LeNet5 and the VGG11 networks, both trained on the Cifar10 dataset. For these experiments, $C = 100 - (Accuracy + 4 * P_i / \sum_{j \in \{all\ layers\}} P_j)$ is used as the cost function, where $Accuracy$ is the estimated accuracy after pruning the ith layer and P_i is the number of parameters in the ith layer. For pruning, x is defined equal to 20, and for fine-tuning during the process, y is defined equal to 2. The results are presented in Figs. 16a and 17a. It can be seen from the figures that the methodology helps maintain the accuracy close to its baseline till a significant

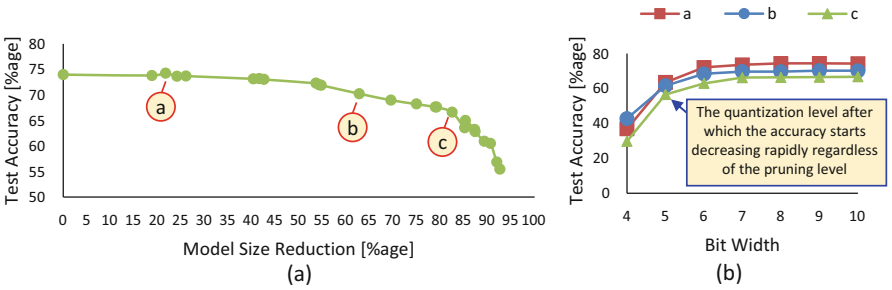


Fig. 16 Results of structured pruning when applied to the LeNet5 network trained on the Cifar10 dataset (Hanif and Shafique 2021). (**a**) Impact of structured pruning on accuracy. (**b**) Impact of quantization on the accuracy of the models having different compression ratios. The models are marked in (**a**)

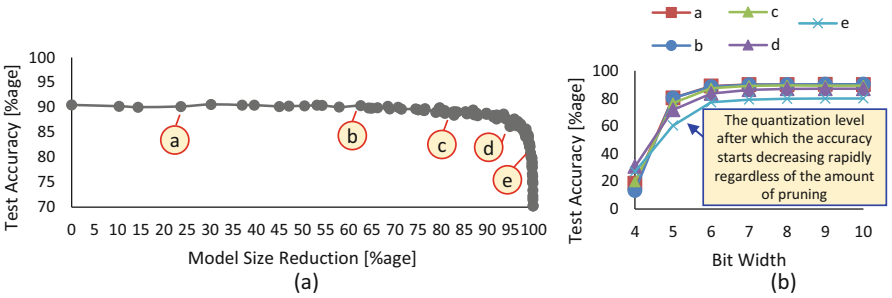


Fig. 17 Results of structured pruning when applied to the VGG11 network trained on the Cifar10 dataset (Hanif and Shafique 2021). (**a**) Impact of structured pruning on accuracy. (**b**) Impact of quantization on the accuracy of the models having different compression ratios. The models are marked in (**a**)

amount of compression, and, after a point, any further compression results in a rapid decrease in the accuracy. Note that intermediate fine-tuning, i.e., $y > 0$, is the key factor for achieving a high compression ratio.

Quantization

To further compress the DNN and to simplify the arithmetic modules in hardware accelerators, network quantization (i.e., Step 2 in Fig. 14) is applied after pruning. For this study, post-training quantization approach with uniform bit-width across the network is considered, for both weights and activations. To quantize the weights of a layer, the following equations are employed:

$$\hat{W_i^{<l>}} = round(W_i^{<l>} \times W_{scale}^{<l>}) \tag{8}$$

$$W_{scale}^{<l>} = 2^{floor(log_2(\frac{2^{n-1}-1}{max(abs(W^{<l>}))}))}$$

where $W^{<l>}$ is the set of all the weights, $W_i^{<l>}$ is the ith element in $W^{<l>}$, $\hat{W^{<l>}}$ represents the set of quantized weights, $W_{scale}^{<l>}$ is the scale factor, and n is the bit-width.

To quantize the activations, first, the activations are profiled using a set of input samples, and then the scale factor is defined using the following equation:

$$A_{scale}^{<l>} = 2^{floor\left(log_2\left(\frac{2^{n-1}-1}{max(abs(A^{<l>}))}\right)\right)}$$

Here $A^{<l>}$ is the set of all the logged activations from the input of the lth layer, and $A_{scale}^{<l>}$ is the scale factor. During the run-time, the activations are scaled with the help of following equation:

$$\hat{A_i^{<l>}} = round(A_i^{<l>} \times A_{scale}^{<l>}) \tag{9}$$

where $\hat{A^{<l>}}$ represents the quantized activations. Note that $W_{scale}^{<l>}$ and $A_{scale}^{<l>}$ are intentionally defined to be in the power of two to simplify the intermediate conversion operations.

Figure 16b shows the accuracies of five DNNs when exposed to different levels of quantization. All the DNNs are variants of the same LeNet5 model trained on the Cifar10 dataset but have different pruning ratios. The baseline models are marked in Fig. 16a with the help of labels. From the figure, it can be observed that the networks with high compression ratios are more sensitive to quantization. Moreover, the accuracy of the networks drops sharply after a specific quantization level. The same trend is observed for the VGG11 network trained on the Cifar10 dataset (see Fig. 17). From this analysis, it can be concluded that higher pruning levels are usually more beneficial than post-training quantization for achieving high overall compression while maintaining close to the baseline accuracy.

Hardware-Level Approximations: Impact of Self-Healing and Nonself-Healing Designs on DNN Accuracy

This section analyzes the impact of using approximate arithmetic modules for internal dot product operations of DNNs on their accuracy. This corresponds to Step 4 in Fig. 14. For this analysis, modules designed using conventional as well as self-healing methods are employed. The key distinction between these designs can be observed from Fig. 18. Figure 18a illustrates a system where the computational modules are replaced with their approximate variants without considering the overall computational flow. In such designs, the selection can be based on thorough design space exploration, but the system is not designed in a manner that the approximation error of one module is compensated by the error of the other modules. The self-healing designs exploit the fact that most of the real-world

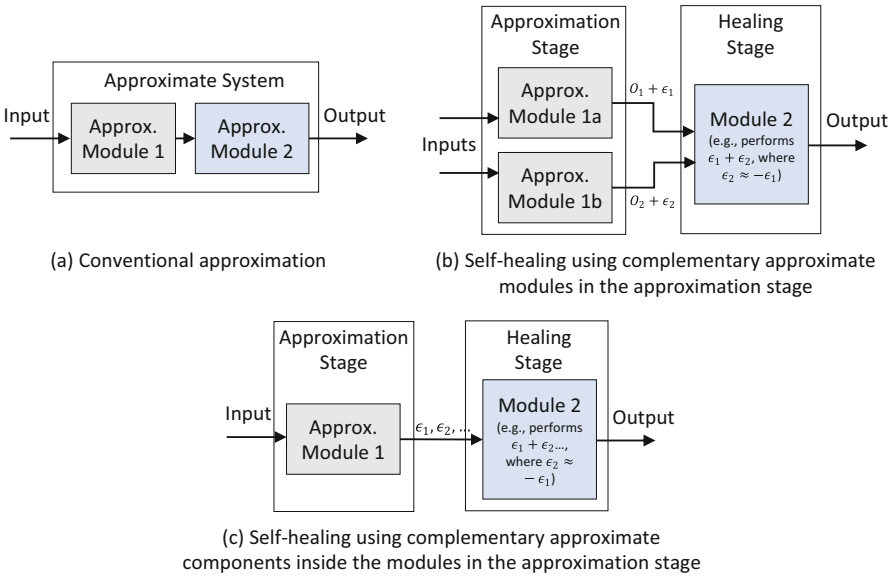


Fig. 18 A comparison of conventional and self-healing approaches (Hanif and Shafique 2021)

systems involve accumulation of multiple computations. The accumulation stage is viewed as the healing stage, while the computational modules are approximated such that they generate complementary errors (Gillani et al. 2018, 2019). This way the error generated by one module is compensated by the error in other modules, and the overall application-level accuracy is not affected. The key advantage of self-healing is that it allows to apply more aggressive approximations in the system compared to the conventional methodology. Figure 18b and c show two different methods for introducing self-healing-based approximations in a system.

The dot product operation is the most common operation involved in DNN execution. It comprises multiplications followed by the accumulation of the products. As multiplication is one of the most costly operations, in this work, approximations are deployed in the multipliers in hardware accelerators. Moreover, conventional as well as self-healing approximate multipliers are considered to study the effectiveness of functional approximations in arithmetic circuits. Figure 19a shows the baseline 8×8 multiplier design used in this work, constructed using 2×2 multipliers. The design of the accurate 2×2 multiplier is shown in Fig. 20a. For approximations, the designs shown in Fig. 20b–20d are employed, where the designs in Fig. 20b and d approximate 3×3 to 7 and 5, respectively (i.e., negative error), and the design in Fig. 20c approximates 3×3 to 11. The 8×8 multiplier configurations used in the analysis are illustrated in Fig. 19b–19j, and their error characteristics are presented in Table 5. Note, for this analysis, it is assumed that the same multiplier design is used for all the multipliers in the hardware accelerator, i.e., homogeneous design. The approximate multiplier configurations that are composed of modules that generate only negative errors represent the conventional multipliers

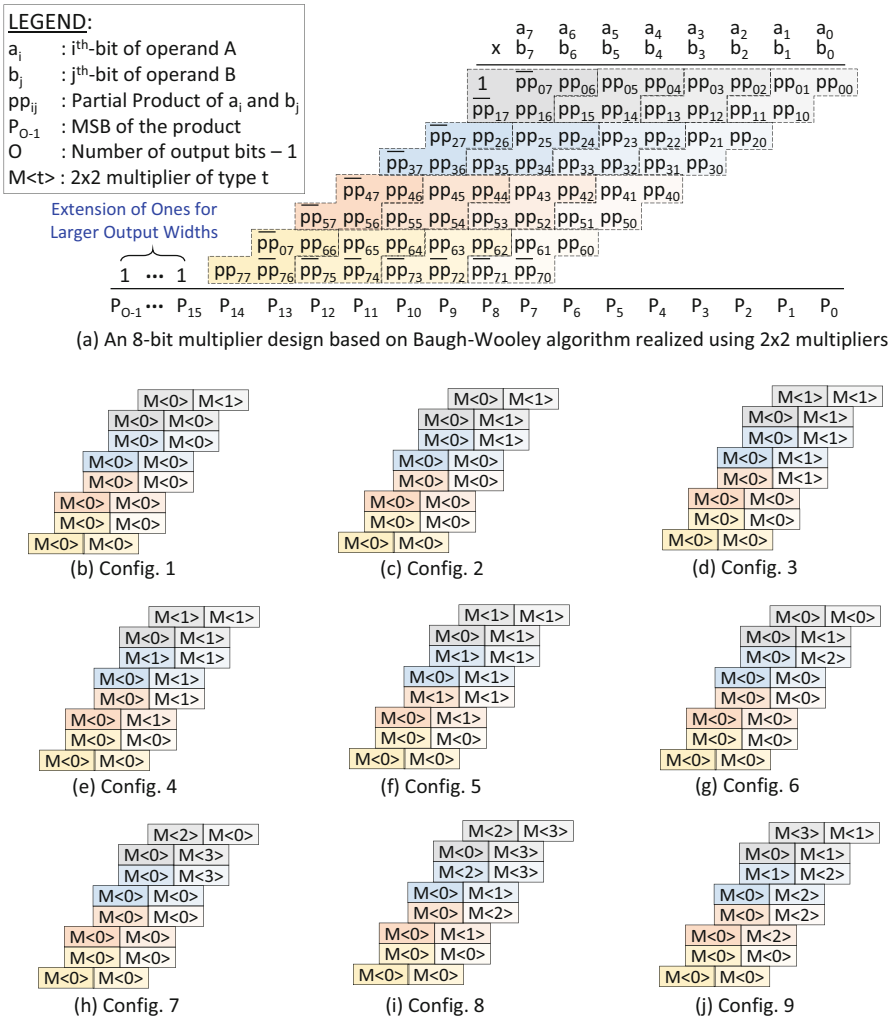
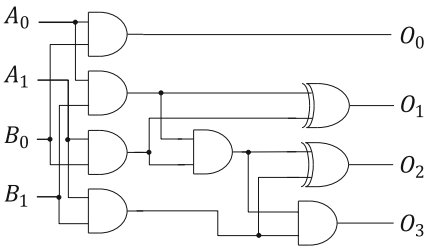


Fig. 19 Types of 8 × 8 approximate multipliers considered for simulations (Hanif and Shafique 2021). (**a**) An 8-bit multiplier design based on Baugh-Wooley algorithm realized using 2 × 2 multipliers. (**b**) Config. 1. (**c**) Config. 2. (**d**) Config. 3. (**e**) Config. 4. (**f**) Config. 5. (**g**) Config. 6. (**h**) Config. 7. (**i**) Config. 8. (**j**) Config. 9

(i.e., configurations in Fig. 19b–19f), and the configurations that generate both positive and negative errors represent the self-healing designs (i.e., configurations in Fig. 19g–19j). The hardware characteristics of all the configurations are presented in Table 6. The results are generated for 65 nm technology using Cadence Genus Synthesis tool with TSMC 65 nm library.

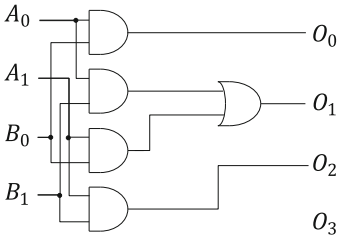
To evaluate the impact of approximations on the accuracy of DNNs, functional models of these approximate multipliers are integrated in a PyTorch-based simulation framework. Figure 21 shows the results obtained when different approximate



(a) Accurate 2x2 multiplier: M<0>

A \ B	00	01	10	11
00	0000	0000	0000	0000
01	0000	0001	0010	0011
10	0000	0010	0100	0110
11	0000	0011	0110	1001

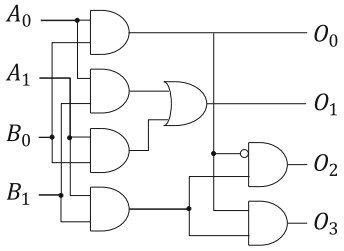
(e) Truth table of M<0>



(b) Approximate 2x2 multiplier having 3x3→7: M<1>

A \ B	00	01	10	11
00	0000	0000	0000	0000
01	0000	0001	0010	0011
10	0000	0010	0100	0110
11	0000	0011	0110	0111

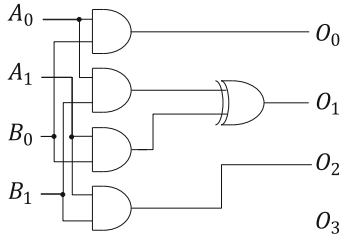
(f) Truth table of M<1>



(c) Approximate 2x2 multiplier having 3x3→11: M<2>

A \ B	00	01	10	11
00	0000	0000	0000	0000
01	0000	0001	0010	0011
10	0000	0010	0100	0110
11	0000	0011	0110	1011

(g) Truth table of M<2>



(d) Approximate 2x2 multiplier having 3x3→5: M<3>

A \ B	00	01	10	11
00	0000	0000	0000	0000
01	0000	0001	0010	0011
10	0000	0010	0100	0110
11	0000	0011	0110	0101

(h) Truth table of M<3>

Fig. 20 The 2 × 2 multiplier designs used for building 8 × 8 approximate multipliers (Hanif and Shafique 2021). (**a**) Accurate 2 × 2 multiplier: M<0>. (**b**) Approximate 2 × 2 multiplier having 3 × 3 → 7: M<1>. (**c**) Approximate 2 × 2 multiplier having 3 × 3 → 11: M<2>. (**d**) Approximate 2 × 2 multiplier having 3 × 3 → 5: M<3>. (**e**) Truth table of M<0>. (**f**) Truth table of M<1>. (**g**) Truth table of M<2>. (**h**) Truth table of M<3>

Table 5 Error characteristics of the multiplier configurations presented in Fig. 19 (Hanif and Shafique 2021)

	Multiplier configurations								
	Ax. 1	Ax. 2	Ax. 3	Ax. 4	Ax. 5	Ax. 6	Ax. 7	Ax. 8	Ax. 9
MSE	0.25	9.75	266.25	3102.30	24806.00	7.50	78.00	2128.00	2547.00
MED	0.13	1.13	7.13	23.13	55.13	0.94	3.38	19.94	21.90
Mean error	-0.13	-1.13	-7.13	-23.13	-55.13	0.00	0.00	-0.25	-0.13

Table 6 Hardware characteristics of the multiplier configurations presented in Fig. 19 (Hanif and Shafique 2021)

	Multiplier configurations									
	Accurate	Ax. 1	Ax. 2	Ax. 3	Ax. 4	Ax. 5	Ax. 6	Ax. 7	Ax. 8	Ax. 9
Area [cell area]	753	716	696	616	609	571	726	727	672	670
Power [μW]	46.04	44.98	44.92	40.81	40.98	38.96	45.49	45.05	43.48	42.94
Delay [ns]	1.92	1.86	1.73	1.73	1.73	1.73	1.95	1.87	1.73	1.77
PDP [fJ]	88.40	83.66	77.71	70.60	70.90	67.40	88.71	84.24	75.22	76.00

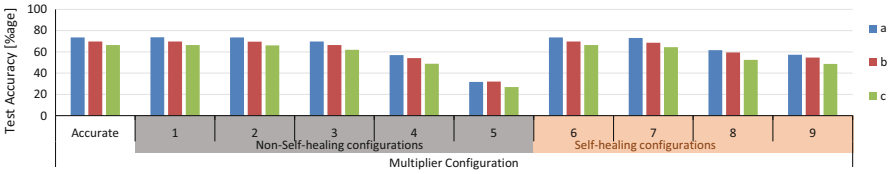


Fig. 21 Impact of using approximate multipliers on the accuracy of different pruned variants of the LeNet5 network (Hanif and Shafique 2021). The considered variants are marked in Fig. 16a

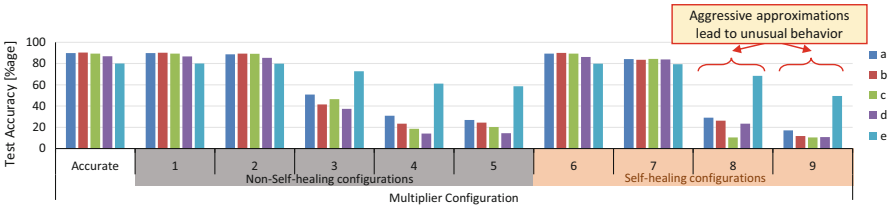


Fig. 22 Impact of using approximate multipliers on the accuracy of different pruned variants of the VGG11 network (Hanif and Shafique 2021). The considered variants are marked in Fig. 17a

multiplier configurations (shown in Fig. 19) are used for the LeNet5 network trained on the Cifar10 dataset. Note, for this analysis, multiple variants of the network are considered, each having experienced a different level of pruning. The network variants are highlighted in Fig. 16a. As can be seen in Fig. 21, with an increase in the compression ratio, the model becomes increasingly sensitive to approximations. Similar results are observed for the case of VGG11 network (see Fig. 22).

Conclusions

Approximations can offer high energy savings while meeting user-defined quality constraints. Besides the well-known techniques such as quantization (i.e., bit-width reduction) and code simplification (e.g., reducing the number of iterations of a loop), it is possible to approximate the functionality of circuits as well. The first part of the chapter primarily focused on functional approximations, where approaches for building approximate components such as adders and multipliers using both manual and automated methods were introduced.

The following section focused on the construction of complex hardware accelerators using existing libraries of approximate components (such as *EvoApproxLib*, *lpAcLib*, or *GeAR*). Two different types of accelerators were presented. For accelerators with irregular structures such as image processing accelerators, an automatic design space exploration and circuit approximation methodology *AutoAx* was presented. This methodology replaces operations in an original accelerator with approximate variants taken from a library of approximate components/circuits. To accelerate the approximation process, QoR and hardware parameters are estimated using computational models created using machine learning methods. It was shown that AutoAx methodology generates approximate accelerators that offer high-quality trade-offs between QoR and hardware parameters. The trade-offs are better than the SoA approaches based on selecting components with the same error or random selection.

The authors also focused on accelerators with a regular structure of processing elements. The methodology *ALWANN* that allows us to approximate hardware accelerators of convolutional neural networks and optimize their energy consumption for inference was introduced. Better energy savings with the same accuracy than the other algorithms that employ retraining were achieved. The retraining typically results in (i) approximation of significantly smaller networks due to scalability issues (Mrazek et al. 2016; Zhang et al. 2015) or (ii) limited set of considered approximate components (Sarwar et al. 2018).

Functional approximation is not the only approach to trade quality for energy efficiency. Developers may also use other techniques such as quantization and pruning. Toward this, a *cross-layer* optimization for neural networks was presented, which systematically combines software-level and hardware-level approximation techniques. The results showed that cross-layer optimization results in better quality-efficiency trade-off. However, note that cross-layer approximate computing is still an active area of research that is yet to uncover the ultimate potential of approximate computing. One of the key hurdles toward achieving that is the lack of sophisticated methodologies for evaluating the error masking and propagation characteristics of approximations, which will enable the projection of approximations across layers and therefore enable fast design space exploration. From approximations for DNNs' perspective, as most of the approximate components have irregular error distribution, there is a need for methodologies to adapt (retrain) DNNs for such approximations. Apart from that, there is a dire need to explore and determine the security of approximated DNNs against adversarial attacks.

Acknowledgments This work was partially supported by the Czech science foundation project 21-13001S.

References

Bailey B, Martin G, Piziali A, Burton M, Greenbaum J, Hashmi K, Haverinen A, Lavagno L, Meredith M, Murray B et al (2007) ESL design and verification: a prescription for electronic system level methodology. Elsevier Science. https://books.google.cz/books?id=raoeAQAAIAAJ

Češka M, Matyáš J, Mrazek V, Sekanina L, Vasicek Z, Vojnar T (2017) Approximating complex arithmetic circuits with formal error guarantees: 32-bit multipliers accomplished. In: 2017 IEEE/ACM international conference on computer-aided design (ICCAD), pp 416–423

Chandrasekharan A, Soeken M, Große D, Drechsler R (2016) Approximation-aware rewriting of AIGs for error tolerant applications. In: Proceedings of the 35th international conference on computer-aided design, ICCAD'16. ACM, New York, pp 83:1–83:8

Chan WTJ, Kahng AB, Kang S, Kumar R, Sartori J (2013) Statistical analysis and modeling for error composition in approximate computation circuits. In: 2013 IEEE 31st international conference on computer design (ICCD), pp 47–53

Chang IJ, Mohapatra D, Roy K (2011) A priority-based 6t/8t hybrid sram architecture for aggressive voltage scaling in video applications. IEEE Trans Circuits Syst Video Technol 21(2):101–112

Chen TH, Alaghi A, Hayes JP (2014) Behavior of stochastic circuits under severe error conditions. it – Inf Technol 56(4):182–191

Chippa VK, Chakradhar ST, Roy K, Raghunathan A (2013) Analysis and characterization of inherent application resilience for approximate computing. In: Proceedings of 50th ACM/EDAC/IEEE design automation conference (DAC), pp 1–9

Choi J, Venkataramani S (2019) Approximate computing techniques for deep neural networks. Springer, Cham, pp 307–329

Deb K, Pratap A, Agarwal S, Meyarivan T (2002) A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans Evol Comput 6(2):182–197

Du K, Varman P, Mohanram K (2012) High performance reliable variable latency carry select addition. In: Proceedings of the conference on design, automation and test in Europe, DATE'12. EDA Consortium, San Jose, pp 1257–1262

Esmaeilzadeh H, Sampson A, Ceze L, Burger D (2012) Architecture support for disciplined approximate programming. In: ACM SIGPLAN notices, vol 47. ACM, pp 301–312

Gillani GA, Hanif MA, Krone M, Gerez SH, Shafique M, Kokkeler AB (2018) Squash: approximate square-accumulate with self-healing. IEEE Access 6:49112–49128

Gillani G, Hanif MA, Verstoep B, Gerez SH, Shafique M, Kokkeler AB (2019) Macish: designing approximate MAC accelerators with internal-self-healing. IEEE Access 7:77142–77160

Gupta V, Mohapatra D, Park SP, Raghunathan A, Roy K (2011) IMPACT: imprecise adders for low-power approximate computing. In: Proceedings of 17th IEEE/ACM international symposium on low-power electronics and Design, pp 409–414

Hanif MA, Shafique M (2021) A cross-layer approach towards developing efficient embedded deep learning systems. In: Microprocessors and microsystems p 103609

Hanif MA, Hafiz R, Hasan O, Shafique M (2017) Quad: design and analysis of quality-area optimal low-latency approximate adders. In: DAC design automation conference 2017. ACM, New York, pp 42:1–42:6

Hashemi S, Tann H, Reda S (2018) BLASYS: approximate logic synthesis using boolean matrix factorization. In: Proceedings of the 55th annual design automation conference, DAC 2018, San Francisco, 24–29 June 2018. ACM, pp 55:1–55:6. https://doi.org/10.1145/3195970.3196001

He K, Zhang X, Ren S, Sun J (2015) Deep residual learning for image recognition. CoRR **abs/**1512.03385

Hrbacek R, Sekanina L (2014) Towards highly optimized cartesian genetic programming: from sequential via simd and thread to massive parallel implementation. In: Proceedings of the 2014 annual conference on genetic and evolutionary computation, GECCO'14. ACM, New York, pp 1015–1022

Jiang H, Liu C, Liu L, Lombardi F, Han J (2017) A review, classification, and comparative evaluation of approximate arithmetic circuits. J Emerg Technol Comput Syst 13(4):60:1–60:34

Judd P, Albericio J, Hetherington T, Aamodt T, Enright Jerger N, Urtasun R, Moshovos A (2018) Proteus: exploiting precision variability in deep neural networks. Parallel Comput 73:40–51

Kulkarni P, Gupta P, Ercegovac M (2011) Trading accuracy for power with an underdesigned multiplier architecture. In: 2011 24th internatioal conference on VLSI design, pp 346–351

Kyaw KY, Goh WL, Yeo KS (2010) Low-power high-speed multiplier for error-tolerant application. In: 2010 IEEE international conference of electron devices and solid-state circuits (EDSSC), pp 1–4

LeCun Y, Bengio Y, Hinton G (2015) Deep Learning. Nature 521(7553):436–444

Li C, Luo W, Sapatnekar SS, Hu J (2015) Joint precision optimization and high level synthesis for approximate computing. In: Proceedings of the 52nd annual design automation conference, DAC'15. ACM, New York, pp 104:1–104:6

Lotfi A, Rahimi A, Yazdanbakhsh A, Esmaeilzadeh H, Gupta RK (2016) Grater: an approximation workflow for exploiting data-level parallelism in FPGA acceleration. In: 2016 design, automation test in Europe conference exhibition (DATE), pp 1279–1284

Lu SL (2004) Speeding up processing with approximation circuits. Computer 37(3):67–73

Ma J, Hashemi S, Reda S (2019) Approximate logic synthesis using blasys. In: Proceedings of 1st workshop on open-source EDA technology (WOSET), p 3

Mahdiani HR, Ahmadi A, Fakhraie SM, Lucas C (2010) Bio-inspired imprecise computational blocks for efficient vlsi implementation of soft-computing applications. IEEE Trans Circuits Syst I: Regul Pap 57(4):850–862

Mazahir S, Hasan O, Hafiz R, Shafique M (2017a) Probabilistic error analysis of approximate recursive multipliers. IEEE Trans Comput 66(11):1982–1990

Mazahir S, Hasan O, Hafiz R, Shafique M, Henkel J (2017b) Probabilistic error modeling for approximate adders. IEEE Trans Comput 66(3):515–530

Mishchenko A, Chatterjee S, Brayton R (2006) Dag-aware aig rewriting: a fresh look at combinational logic synthesis. In: 2006 43rd ACM/IEEE design automation conference, pp 532–535

Mishra AK, Barik R, Paul S (2014) iact: a software-hardware framework for understanding the scope of approximate computing. In: Workshop on approximate computing across the system stack (WACAS)

Mohapatra D, Chippa VK, Raghunathan A, Roy K (2011) Design of voltage-scalable meta-functions for approximate computing. In: Design, automation & test in Europe conference & exhibition (DATE), 2011. IEEE, pp 1–6

Momeni A, Han J, Montuschi P, Lombardi F (2015) Design and analysis of approximate compressors for multiplication. IEEE Trans Comput 64(4):984–994

Mrazek V, Sarwar SS, Sekanina L, Vasicek Z, Roy K (2016) Design of power-efficient approximate multipliers for approximate artificial neural networks. In: Proceedings of the 35th international conference on computer-aided design, ICCAD'16. ACM, New York, pp 81:1–81:7

Mrazek V, Hrbacek R, Vasicek Z, Sekanina L (2017) Evoapprox8b: library of approximate adders and multipliers for circuit design and benchmarking of approximation methods. In: Design, automation test in Europe conference exhibition (DATE), 2017, pp 258–261

Mrazek V, Vasicek Z, Hrbacek R (2018) The role of circuit representation in evolutionary design of energy-efficient approximate circuits. IET Comput Digit Tech 12(4):139–149

Mrazek V, Hanif MA, Vasicek Z, Sekanina L, Shafique M (2019a) AutoAx: an automatic design space exploration and circuit building methodology utilizing libraries of approximate components. In: Proceedings of the 56th annual design automation conference 2019. Association for Computing Machinery, New York

Mrazek V, Vasicek Z, Sekanina L, Hanif MA, Shafique M (2019b) ALWANN: automatic layer-wise approximation of deep neural network accelerators without retraining. In: 2019 IEEE/ACM international conference on computer-aided design (ICCAD), pp 1–8

Nair R (2014) Big data needs approximate computing: technical perspective. Commun ACM 58(1):104–104

Nepal K, Li Y, Bahar RI, Reda S (2014) Abacus: a technique for automated behavioral synthesis of approximate computing circuits. In: 2014 design, automation test in Europe conference exhibition (DATE), pp 1–6

Nepal K, Hashemi S, Tann H, Bahar RI, Reda S (2017) Automated high-level generation of low-power approximate computing circuits. In: IEEE transactions on emerging topics in computing, p 1

Pinos M, Mrazek V, Sekanina L (2021) Evolutionary neural architecture search supporting approximate multipliers. In: Hu T, Lourenço N, Medvet E (eds) Genetic programming. Springer International Publishing, Cham, pp 82–97

Ranjan A, Raha A, Venkataramani S, Roy K, Raghunathan A (2014) Aslan: synthesis of approximate sequential circuits. In: 2014 design, automation test in Europe conference exhibition (DATE), pp 1–6

Sarwar SS, Venkataramani S et al (2018) Energy-efficient neural computing with approximate multipliers. J Emerg Technol Comput Syst 14(2):16:1–16:23

Sengupta D, Snigdha FS et al (2017) Saber: selection of approximate bits for the design of error tolerant circuits. In: Design automation conference (DAC)

Shafique M, Ahmad W, Hafiz R, Henkel J (2015) A low latency generic accuracy configurable adder. In: Proceedings of annual design automation conference, DAC'15, pp 86:1–86:6

Shafique M, Hafiz R, Rehman S, El-Harouni W, Henkel J (2016) Cross-layer approximate computing: from logic to architectures. In: Proceedings of 53rd IEEE/ACM design automation conference

Sidiroglou-Douskos S, Misailovic S, Hoffmann H, Rinard M (2011) Managing performance vs. accuracy trade-offs with loop perforation. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on foundations of software engineering. ACM, pp 124–134

Soeken M, Große D, Chandrasekharan A, Drechsler R (2016) BDD minimization for approximate computing. In: Proceedings of the 21st Asia and South Pacific design automation conference (ASP-DAC), pp 474–479

Srinivasan G, Wijesinghe P, Sarwar SS, Jaiswal A, Roy K (2016) Significance driven hybrid 8t-6t sram for energy-efficient synaptic storage in artificial neural networks. In: 2016 design, automation test in Europe conference exhibition (DATE), pp 151–156

Tung F, Mori G (2018) CLIP-Q: deep network compression learning by in-parallel pruning-quantization. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp 7873–7882

Vasicek Z, Sekanina L (2014) How to evolve complex combinational circuits from scratch? In: 2014 IEEE international conference on evolvable systems, pp 133–140

Vasicek Z, Sekanina L (2015) Evolutionary approach to approximate digital circuits design. IEEE Trans Evol Comput 19(3):432–444

Vasicek Z, Mrazek V, Sekanina L (2017) Towards low power approximate DCT architecture for HEVC standard. In: Design, automation test in Europe conference exhibition (DATE), 2017, pp 1576–1581

Vasicek Z, Mrazek V, Sekanina L (2019) Automated circuit approximation method driven by data distribution. In: 2019 design, automation test in Europe conference exhibition (DATE), pp 96–101

Vaverka F, Mrazek V, Vasicek Z, Sekanina L, Hanif MA, Shafique M (2020) Tfapprox: towards a fast emulation of dnn approximate hardware accelerators on GPU. In: 2020 design, automation and test in Europe conference (DATE), p 4

Venkataramani S, Sabne A, Kozhikkottu V, Roy K, Raghunathan A (2012) Salsa: systematic logic synthesis of approximate circuits. In: DAC design automation conference 2012, pp 796–801

Venkataramani S, Roy K, Raghunathan A (2013) Substitute-and-simplify: a unified design paradigm for approximate and quality configurable circuits. In: 2013 design, automation test in Europe conference exhibition (DATE), pp 1367–1372

Venkataramani S, Ranjan A, Roy K, Raghunathan A (2014) Axnn: energy-efficient neuromorphic systems using approximate computing. In: 2014 IEEE/ACM international symposium on low power electronics and design (ISLPED), pp 27–32

Xu Q, Kim NS, Mytkowicz T (2016) Approximate computing: a survey. IEEE Des Test 33(1): 8–22

Zhang Q, Wang T, Tian Y, Yuan F, Xu Q (2015) Approxann: an approximate computing framework for artificial neural network. In: Design, automation test in Europe conference exhibition (DATE), pp 701–706

Zhu N, Goh WL, Yeo KS (2009) An enhanced low-power high-speed adder for error-tolerant application. In: Proceedings of the 2009 12th international symposium on integrated circuits, pp 69–72

Parallel Programming Models

30

Muhammad Nufail Farooqi, Mustafa Abduljabbar, Vicenç Beltran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Miquel Pericàs

Contents

Introduction . . . 1070
Hardware Models . . . 1070
Constructs in Parallel Programming Models . . . 1071
Taxonomy . . . 1072
The OpenMP Programming Model . . . 1074
The Worksharing Model . . . 1076
The Tasking Model . . . 1077
SIMD Support in OpenMP . . . 1079
The Accelerator Model . . . 1085
The OmpSs-2 Programming Model . . . 1087
Advanced Dependency System . . . 1087
Exploiting Structured Parallelism on Many-Core Processors . . . 1091
OmpSs-2 NUMA Support . . . 1093
The XiTAO Programming Model and Runtime . . . 1096
Explicit DAG Programming in XiTAO . . . 1096
Software Topologies and Locality-Aware Programming . . . 1098

M. N. Farooqi
Leibniz Supercomputing Centre of the Bavarian Academy of Sciences and Humanities, Munich, Germany
e-mail: farooqi@lrz.de

M. Abduljabbar
The Ohio State University, Columbus, USA
e-mail: abduljabbar.1@osu.edu

M. Pericàs (✉)
Chalmers University of Technology, Gothenburg, Sweden
e-mail: miquelp@chalmers.se

V. Beltran · X. Teruel · R. Ferrer · X. Martorell
Barcelona Supercomputing Center, Barcelona, Spain
e-mail: vbeltran@bsc.es; xavier.teruel@bsc.es; roger.ferrer@bsc.es; xavier.martorell@bsc.es

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_30

The XiTAO Data-Parallel Interface 1100
The XiTAO Runtime 1102
Conclusion 1103
References 1104

Abstract

Parallel systems have evolved to become highly hierarchical and heterogeneous, requiring tools to program at the level of clusters, nodes, vectors, and accelerators such as GPUs and FPGAs. This chapter describes existing methodologies to program across these multiple levels and then proceeds to describe in detail the features of three modern parallel programming environments that focus on node and SIMD level parallelism, namely OpenMP, OmpSs-2, and XiTAO. The chapter starts by presenting a taxonomy of parallel programming models and then describes the three focus programming models in sequence.

Keywords

Parallel programming models · Accelerators · Tasking · SIMD · OpenMP · OmpSs · XiTAO

Introduction

Parallel Programming Models provide a way for programmers to write parallel programs that can express the inherent parallelism in the problem domain and execute these programs efficiently on the underlying hardware. Both hardware and programming models are evolving rapidly, thereby adding new features to hardware that increase performance and productivity. Some knowledge of hardware models and related terminology is useful to understand parallel programming models and its constructs. A summary of hardware models used in parallel computer architectures and constructs for expressing the parallelism is defined before introducing the taxonomy of parallel programming models.

Hardware Models

The evolution of hardware models greatly influenced the parallel programming models that are available today. Memory and computational models are the two most important factors of hardware in defining most of the programming models. In current architectures there exist two memory models, shared memory and distributed memory, that dictate how communication is handled among parallel processes during execution.

In shared memory architectures, a compute node is comprised of one or multiple processors such that each processor is connected to a local memory. While the cores in a processor usually observe the same latency when going to memory (known

as uniform memory access, UMA), in a multiprocessor setting, a processor can access its local memory with lower latency and remote memory attached to other processors with a higher latency. This is then known as non-uniform memory access (NUMA). The value of latency is dependent upon the distance between the memory where the data resides and the processor accessing that data. The performance of parallel programs can drastically reduce when the NUMA effect is not taken into account when performing data access.

Because it is difficult to put a huge number of processors/cores on a single node, supercomputers/computing clusters are built by connecting a large number of nodes together through high-speed interconnect. Each compute node consists of its own private memory; therefore, a processor/core in one compute node cannot directly access memory in another compute node. To solve this issue, an indirect access can be provided through software; this arrangement is known as distributed memory architecture.

Computational requirements of parallel programs vary from one problem domain to another. This variability in computational requirements is addressed by putting together many computational units of the same and/or varying compute capability.

A compute node where many of the same computational units are put together is known as homogeneous architecture. On the other hand, when computational units with different compute capabilities and/or instruction sets are put together, they are known as heterogeneous architectures. The heterogeneous systems where the difference among computational units is their compute capability only are known as asymmetric multiprocessing or heterogeneous multiprocessing (HMP) architectures. An example of such architecture is Arm's big.LITTLE.

The term heterogeneous system, however, is commonly used for computer systems where computational units with different instruction sets and compute capability are put together, e.g., compute nodes with accelerators. The accelerators in a heterogeneous system can share the memory with the CPU or can have their own private memory. Accelerators that share the memory with the processors are known as tightly coupled accelerators, e.g., vector units and on-chip GPU/FPGA, and accelerators that have their own memory are categorized as loosely coupled accelerators.

Constructs in Parallel Programming Models

Parallel Programming Models use a number of abstractions/constructs to express parallelism in a program that helps in efficient execution/mapping on the underlying parallel hardware. These abstractions enable portability as compared to programming directly for a specific hardware model.

Parallel programming models usually support two types of parallelism, i.e., data and task parallelism. In data parallelism, a single program/instruction running on multiple processing units is applied to a separate set of data, e.g., a parallel loop. Commonly used abstractions for data parallelism are SIMD (Single Instruction, Multiple Data), SPMD (Single Program, Multiple Data), and vectors. In theory, SIMD and vectors both refer to the execution of a single instruction on multiple

data; however, their implementations in hardware are different. A SIMD unit, comprised of multiple processing units, executes an instruction on an array of data simultaneously while a vector unit executes an instruction on an array of data in a pipeline fashion. Lastly, a program or task running in multiple processes or threads simultaneously, on different processors and working on a disjoint set of data, is known as SPMD.

A task is an independent unit of work that can be executed by a process or a thread depending on the programming model. Multiple tasks running concurrently on multiple processors are known as task parallelism. There exist many parallel programming models that provide support for task parallelism. Some of these programming models are discussed later in this chapter in detail. These models provide means for programmers to define tasks along with their dependencies and later execute these tasks efficiently on the underlying hardware.

Besides the types of parallelism, programming models provide a number of patterns and features that help programmers to express commonly used operations in parallel programming, for example, the fork-join pattern (Swamy 2012), synchronization primitives, atomic operations, and reduction operations. In the fork-join pattern, a number of threads/processes are spawned/forked in parallel to run either the same or a different set of instructions and merge/join together at the end. The fork-join pattern is mostly used to support irregular parallel programming patterns; however, regular programming patterns, e.g., however, regular programming patterns such as 'OpenMP parallel for' are also implemented using it. Synchronization primitives are used to halt the execution of threads/processes at a certain point in a program, as well as to regulate access to critical region of code. Atomics make sure that a data element can only be modified by a single process/thread at a time. Reduction operations tell the programming models that it is to be applied on an array of elements, and results are accumulated in the specified reduction variable.

Programs running on heterogeneous machines may need to be written in more than one programming model, where each programming model targets a specific computational unit, thus resulting in multiple source codes. New programming models are developed, and some existing models are evolved to eliminate the need for multiple source codes and rather use a single source code where the same source code (with the help of compilers) is compiled for each target computational unit.

Taxonomy

Parallel programming models are highly dependent on underlying hardware and are usually developed for specific hardware. Two main hardware features that provide a basis for programming models' classification are memory and heterogeneity in compute hardware. On the basis of memory, parallel programming models can be classified into three types: shared memory models, distributed memory models, and distributed shared memory models. While based on heterogeneity of compute hardware, accelerator models were developed. A hybrid approach can be taken to combine some of these models depending on the machine model. Figure 1 shows a

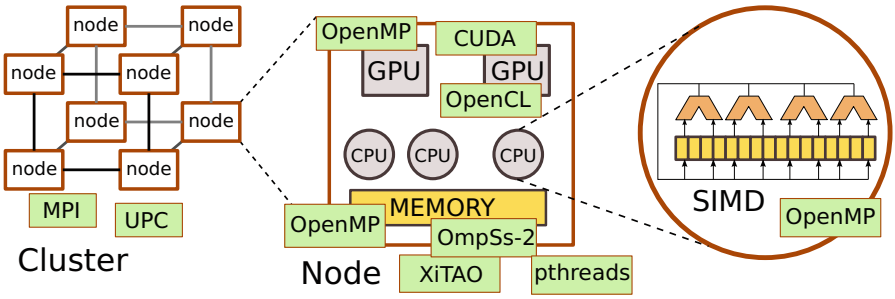


Fig. 1 Taxonomy of parallel programming models showing the levels of a parallel computer system and associated programming models highlighted in green boxes

taxonomy of the parallel programming models with examples. Next, these models are described separately.

Shared memory models: In a shared memory model, processes/threads running in parallel can communicate directly via the physical memory by allowing them simultaneous access to the data. Shared memory provides a fast way for communication; however, simultaneous access to data can lead to race conditions. To handle this issue, access to shared data is regulated using synchronization mechanisms for data consistency. OpenMP, OmpsS, XiTAO, and Pthreads are examples of shared memory programming models. This chapter describes in sections "The OpenMP Programming Model", "The OmpSs-2 Programming Model", and "The XiTAO Programming Model and Runtime" the shared memory parallel programming models in detail.

Distributed memory models: These models are used to write programs for execution on distributed memory systems where data among processes is communicated through synchronous/asynchronous messages. Unlike shared memory models, distributed memory models solve the problem of data races and can scale beyond a single node, thus making it suitable for large problems that can't fit in a single machine. Message Passing Interface (MPI) is an example of a distributed memory model.

Distributed shared memory models: Also known as Partitioned Global Address Space (PGAS) models, combine the ease of programming of shared memory models with the scalability of message passing models by providing an abstraction layer above the two models. Unified Parallel C and Coarray FORTRAN are examples of PGAS models.

Heterogeneous models: These are the programming models for accelerators. Most of the programming models in this category are developed by hardware vendors that are specific for their devices, e.g., CUDA, OpenCL, and SYCL.

Hybrid models: Depending on the machine model, multiple programming models can be combined together and provide the opportunity to combine the advantages of these models. For example, OpenMP + CUDA are combined for machines with multicore CPUs and GPUs. The OpenMP + MPI combination is used in programs that are targeted to run on clusters, whereby OpenMP is used within a node and MPI is used across nodes.

The OpenMP Programming Model

OpenMP has become a de facto standard for parallel programming in shared memory environments, although its latest versions also contemplate the offload-to-device model, which can also work with distributed memory address spaces. In 1997 appeared its first release (OpenMP Architecture Review Board 1997) having Fortran as its initial base language. The main objective was to allow the parallelization of codes written by specific domain scientists who did not have a particular knowledge of parallel computer architectures. A version for C/C++ would soon appear (OpenMP Architecture Review Board 1998) to extend the scope of its usage. These two versions of the standard (Fortran and C/C++) coexisted separately in successive releases until the appearance of OpenMP 2.5 (OpenMP Architecture Review Board 2005), where both of them were compiled into a single document. Since that release, the document is focused on defining the semantics of the model generically, opening specific blocks for the syntax of each of the supported base languages, as well as clarifying the particularities that exclusively affect one of them.

The main idea of OpenMP is based on the use of three different components: (a) directives; (b) environment variables; and (c) library routines. Directives are source code annotations that allow the preprocessor to inject code at specific points in the user code. The directives are associated with certain semantics that programmers want to include in their codes. For instance, if a programmer adds the `critical` directive annotating a given code block, they are indicating the need to execute that code block in mutual exclusion. On the other hand, environment variables allow configuring the behavior of the program at runtime. Then, if the programmer initializes the environment variable `OMP_NUM_THREADS` before executing the program, they are indicating the number of threads they want to use for that particular execution. Finally, the library routines allow to configure (or to request) the current status of the program. This way, the programmer can get the number of threads used in a particular region by calling the `omp_get_num_threads()` OpenMP routine.

Throughout its history, OpenMP has developed a set of sub-models that allow programmers to parallelize their applications using different approaches. From the beginning until version 2.5, the model was based exclusively on *work-sharing* techniques, that is, distributing the workload among the threads available at runtime. OpenMP 3.0 (OpenMP Architecture Review Board 2008) introduced the *tasking* model, allowing the parallelization of non-well-structured codes that are not easily parallelizable using work-sharing. Finally, OpenMP 4.0 (OpenMP Architecture

Review Board 2013) introduced the *offloading* model, allowing the programmers to take advantage of the accelerator devices attached to the SMP architecture. All of these additions improved the way programmers can express the inherent parallelism of their application. The current version, i.e., OpenMP 5.x (OpenMP Architecture Review Board 2020), has not included any new sub-model per se, but it improves certain features of the existing ones. Also, the major contribution of the latest version is based on the standardization of the interaction between the OpenMP runtime and third-party tools, including performance and debugging tools.

The OpenMP execution model is based on the creation of successive parallel regions separated by sequential regions. That is, it allows to annotate code regions that will be executed by multiple threads, while non-annotated code regions will be executed only by the initial thread. Within a parallel region, programmers can also define certain code subregions that can distribute the associated work among the multiple threads participating in it. Perhaps one of the most used OpenMP features is the `for` directive (`do` in Fortran); this directive allows dividing the iteration space associated with a loop among the different threads participating in the region.

Additionally, a parallel region can also be used for task execution. The specification defines safe points in the execution of the program where threads can look for new work to execute from the ready task pool. A fairly common use case can create a parallel region with multiple threads, restrict the execution to a single thread, and instantiate multiple tasks. Although the encountered code executes with only one thread, any member of the team could be a candidate to execute the instantiated tasks. We can refer to this scenario as one producer, i.e., one thread creates all the tasks, and multiple consumers, i.e., all threads participating in the parallel region execute these tasks.

Apart from these two programming sub-models, both based on the fork-join design pattern, OpenMP additionally offers a complementary pattern that allows code vectorization using the SIMD directives. Although this parallel approach can exist on its own, it is usually combined with other directives allowing the loop parallelization. Thus, the first level of parallelism divides the work between threads, while a second level exploits the internal parallelism offered by the vector instructions of the processor.

Once the OpenMP execution model has been introduced, it is necessary to pay attention to its memory model. OpenMP defines a shared memory model of relaxed consistency. This means that all the threads can access the whole memory space (at least the host memory space; this situation changed once introduced the offload sub-model) and read or write any shared variable. Even so, the model offers us a "relaxed" consistency. This means that not all threads see or make visible the changes they do at any time but at specific points of the executions. To guarantee consistency, the model defines the *flush* operations (also related to the `flush` directive), and it includes this operation implicitly in those constructs that may require it. For example, in the execution of an OpenMP barrier, programmers have the guarantee that all the changes made by the different threads of the team are visible to the rest of the members (and, therefore, that any thread can see the changes made by the rest of the team). The definition of these flushes facilitates

the programming of the application, making the use of this model transparent (in most cases) to the programmer.

The memory model changes when using devices (i.e., in the offload sub-model). In these cases, it will be the programmer who will define the correct order, in terms of data movements, between the host and the device, and vice versa. For this purpose, the `map` clause is available in the language. Examples of its use will be shown in section "The Accelerator Model".

OpenMP has been adopted by a great variety of C, C++, and Fortran compilers, from the major industry/commercial compilers, like Intel, IBM, Fujitsu, Texas Instruments, or Oracle; to other pure open-source approaches, like the GNU Compiler Collection, Mercurium, Rose, or OpenUH compilers. In recent years, the emergence of hybrid approaches (between open-source and commercial initiatives) of the LLVM-based compilers has also been observed, like AMD, Clang, Flang, or PGI. Some of the compilers mentioned above have academic purposes and are used as rapid prototyping mechanisms for the study of new proposals for the standard. This is the case of the Mercurium compiler, used for the implementation of the OmpSs/OmpSs-2 programming model (see section "The OmpSs-2 Programming Model").

Concerning the HPC community, the standard has been adopted in many different ways: from programming exclusively using OpenMP for small problem size workloads (still requiring multi-threaded capabilities) to hybrid approaches in the form of MPI + OpenMP, where it is used as a second level of parallelization, exploiting the intra-node parallelism. Since its version 4.0, it is also considered as a programming mechanism for accelerator devices attached to the host.

Listing 1 OpenMP loop work-sharing construct example (the axpy algorithm)

```
1 #pragma omp parallel
2 {
3 #pragma omp for schedule(dynamic, 256)
4   for (i=0; i<SIZE;i++) {
5     A[i] = a * B[i] + A[i];
6   }
7 }
```

The following sections provide an overview of the four programming sub-models that have just been introduced: the work-sharing model, the tasking model, the SIMD model, and the accelerator (or offload) model.

The Worksharing Model

The OpenMP model introduced the work-sharing mechanism as the original method of parallelization. Its fundamentals are simple: the language allows the creation of a parallel region and then the programmer may decide to distribute the work associated with a given subregion.

Listing 1 shows a simple example of this usage. The first line of the source code defines the parallel region (surrounded by curly brackets), while the second *pragma* (line 3) announces the upcoming work-sharing region. The annotated loop will be divided into portions of 256 iterations (chunk-size parameter), and the OpenMP

runtime will dynamically assign each of them among all the threads participating in the parallel region (schedule policy). Lines 4–6 contain the original user code.

A second parallelization approach is possible when the code has several blocks that can be executed concurrently. In this case, there is not an iteration space that can be distributed among threads, but each of these blocks can still be assigned to a different thread exploiting the program parallelism.

Listing 2 shows a region of code including function calls that can potentially be executed in parallel. The annotation of the code begins with the opening of the parallel region (line 1), the opening of the context of the `sections` construct (line 3), and the demarcation of each of these sections (lines 5, 7, and 9).

Listing 2 OpenMP sections construct

```
1  #pragma omp parallel
2  {
3  #pragma omp sections
4      {
5  #pragma omp section
6          code_blk_1();
7  #pragma omp section
8          code_blk_2();
9  #pragma omp section
10         code_blk_3();
11     }
12 }
```

Listing 3 OpenMP tasking construct

```
1  #pragma omp parallel
2  {
3  #pragma omp single
4      {
5  #pragma omp task
6          code_blk_1();
7  #pragma omp task
8          code_blk_2();
9  #pragma omp task
10         code_blk_3();
11     }
12 }
```

The sections construct offers the possibility of expressing parallelism based on code blocks, but it has two limitations that do not allow to accommodate its use in more complicated scenarios. On the one hand, the number of sections must be known at compile time, which makes it difficult to use it in codes where the number of blocks is variable. On the other hand, each of the sections must be located in the scope of the sections construct, which prevents creating sections within the function calls listed in the previous code.

The Tasking Model

The previous section has shown the parallelization of code blocks using OpenMP. This section still focuses on task parallelism but uses task-related directives. As will be shown, OpenMP tasks are a generalization of the work-sharing sections construct providing greater expressiveness and interesting semantics to use in parallel codes.

A task is a work unit that contains (1) user code; (2) the data associated with that code; and (3) a set of internal variables that will determine its behavior. In a way, programmers can use tasks to implement code where sections were used. The code in Listing 3 shows a direct translation of the code that was previously shown in Listing 2. In this new example, the parallel region is still present, but instead of using the work-sharing `sections` construct, a `single` directive is inserted. It means that just one thread will take care of creating all the tasks. It is important to note that although one single thread creates tasks, all the threads participating in the team will collaborate in its execution. The parallelization of the source code is completed by including the task annotations (lines 5, 7, and 9).

Coming back to one of the restrictions mentioned before, *the number of sections should be known at compile time*. This restriction does not apply when using tasks. The code in Listing 4 shows the initialization of a list of elements by using a while loop (lines 7–12). In this case, loop work-sharing cannot be used (since it requires the canonical form of a `for` loop, where the number of iterations must be known at the entry point), and it is also not possible to use the `sections` construct, due to the restriction previously mentioned. Using tasks, however, is a fairly common and natural pattern for this case.

Listing 4 The taskwait directive

```
1  #pragma omp parallel
2  {
3  #pragma omp single
4    {
5      list_t list = begin;
6
7      while (list != NULL) {
8  #pragma omp task
9        initialize(list->elem);
10
11       list = list->next;
12     }
13
14 #pragma omp taskwait
15
16     list = begin;
17
18     while (list != NULL) {
19 #pragma omp task
20       compute(list->elem);
21
22       list = list->next;
23     }
24   }
25 }
```

Listing 5 Task dependences

```
1  #pragma omp parallel
2  {
3  #pragma omp single
4    {
5      list_t list = begin;
6
7      while (list != NULL) {
```

```
 8 #pragma omp task depend(out: list)
 9       initialize(list->elem);
10
11       list = list->next;
12     }
13
14
15
16     list = begin;
17
18     while (list != NULL) {
19 #pragma omp task depend(inout: list)
20       compute(list->elem);
21
22       list = list->next;
23     }
24   }
25 }
```

Tasks also require certain synchronization points. In the example above, to start the second `while` loop, the *initialization* tasks must have completed. This is achieved with the `taskwait` directive (line 14), which allows to wait for all the tasks created up to that moment.

Despite the correct use of the `taskwait` directive, OpenMP offers other mechanisms that allow defining the correct order of task executions: dependencies. The `depend` clause allows to annotate a task according to the data it is using. In this way, the OpenMP runtime will take care of computing all the possible data dependencies between them. The code in Listing 5 uses this new synchronization approach. The code does not need the `taskwait` directive anymore. The corresponding annotations on the task side have been added (lines 8 and 19). This modification not only guarantees the correct order but also achieves a finer grain of synchronization. That is, to compute an element of the list (second `while` loop, line 19), it is not required that the initialization of all the elements in the list (first `while` loop) has completed, but it is enough to wait only for the element that is currently being computed.

The depend clause supports several types of modifiers. The most common are `in` (read), `out` (write), and `inout` (update), which allow the usual compute of dependencies: RaW (*Read-after-Write*), WaR (*Write-after-Read*), and WaW (*Write-after-Write*).

SIMD Support in OpenMP

SIMD (Single Instruction, Multiple Data) parallelism is a form of data parallelism that does not implement concurrency using threads. Instead SIMD parallelism refers to the usage of SIMD instructions. Modern instruction sets provide instructions that can operate on multiple data elements; these are broadly referred to as SIMD instructions. Operating on multiple data elements allows a program to use fewer instructions to perform the same computation. This reduction of number of instruction is beneficial because modern CPUs can execute SIMD instructions with latencies similar to instructions that operate on single elements.

Listing 6 Examples of data parallelism in source code

```
// Loop with no loop-carried data dependences. Any vector length is useable.
double y[], x[], a;
for (int i = 0; i < N; i++) {
  int w = 2*i;
  y[i] += a*x[i] + w;
}
// Loop with loop-carried data dependences but still vectorizable
// with vectors with a length no larger than 5 elements.
double a[];
for (int i = 0; i < N - 5; i++) {
  a[i] = a[i] + a[i + 5];
}
// Straight line code.
struct pixel { float r, g, b, a; } pixel = ...;
pixel.r *= 0.7;
pixel.g *= 0.3;
pixel.b *= 0.4;
pixel.a *= 0.9;
```

Vectorization, Intrinsics, and Semi-automatic Vectorization

Regular instructions of a CPU operate on *scalar* values, that is, individual values like integers or floating point numbers. SIMD instructions operate on *vectors* of scalar *elements*. From this terminology arose the concept of *vectorization* which is the process by which optimizing compilers can use machine instructions that operate on vectors to compile code that operates on scalars. Vectors have a number of elements, often called the vector *width* or *length*. The typical computation of a SIMD instruction is to apply the same operation to all the elements.

Code that exposes *data parallelism* is often amenable for vectorization. These come mainly in two forms: loops that expose data parallelism (either due to the nature of its memory access patterns or because of the nature of the programming model, such as an OpenCL kernels) and straight-line code that explicitly accesses different kinds of data but performs the same operation to them. Loop vectorization is the usual use case of vectorization as it is where the bulk of the computation happens in almost all applications. For this reason it has extensively been studied in the literature (Kennedy and Allen 2001; Karrenberg 2015). Straight-line code that can be vectorized has been called *Superword Level Parallelism* (Larsen and Amarasinghe 2000), and while it has more limited applicability, it is still useful in some contexts. Listing 6 shows a typical example of data parallelism.

For vectorization to be correct, it requires the compiler to perform a careful analysis of dependences so the semantics of the original program is preserved. This means that, traditionally, the ability of a user to exploit vector instructions depends completely on the quality of implementation of the vectorizer in the compiler. A compiler may have difficulties proving facts on the source code that are actual requirements for a legal vectorization. When this happens, vectorization cannot be applied and no performance improvements are observed. This is a common issue with fully automated vectorization. Compilers can inform the user what prevents the vectorization via some form of compiler reports. Rewriting the source code can help in some circumstances, but it offers no guarantees.

To address this issue, compiler vendors offer different options. One option is a set of annotations so the user can state the facts that the compiler cannot prove. These annotations come in the form of vendor-specific pragma constructs or `assume` intrinsics. Via the reports of the compiler, a user can identify what is actually preventing vectorization. If the fact is true but cannot be proved by the compiler, some form of annotation can enable vectorization again. This is a form of semi-automated vectorization on which the bulk of the vectorization is still handled by the compiler. For instance, it is possible to annotate the alignment of a memory buffer as a fact to be used by the compiler. Some architectures impose memory alignment constraints, or can use more efficient instructions, when loading or storing vectors from or to memory. Annotations can be pushed further so the user can fully override correctness checks. For instance, the user can state that it is correct to vectorize a loop, irrespective of the actual fact. In general, these kinds of annotations are often not portable between compilers.

Another option is to use architecture-specific compiler intrinsics. Those give the maximum control to the user but at a higher programming effort. They are in practice closer to assembly programming because those intrinsics often offer a one-to-one correspondence to the corresponding SIMD instruction. This is a form of manual vectorization as the compiler offers little assistance and the user needs to take all the decisions. Hardware vendors push for standardized low-level intrinsics so they can be used in different compilers, but those intrinsics are obviously non-portable between architectures.

Listing 7 is an example of a SAXPY kernel. And Listing 8 is an example of its manual vectorization using RISC-V Vector Extension intrinsics (RISC-V Community 2021). Compiler intrinsics provide a lot of control. This example shows how the programmer has to make explicit a number of low-level details when using intrinsics: vector loads (`vle32_v_f32m8`), vector stores (`vse32_v_f32m8`), and even a contraction of the addition and multiplication using a specific float-multiply operation (`vfmacc_vf_f32m8`). Another detail of the RISC-V architecture that pervades the code when using intrinsics is the need to request a vector length (`vsetvl_e32m8`) that is then passed to all the intrinsics and also used to advance the loop.

Listing 7 SAXPY kernel

```
void saxpy_scalar(size_t n,
                  const float a,
                  const float *x,
                  float *y) {
 for (size_t i = 0; i < n; ++i) {
  y[i] = a * x[i] + y[i];
 }
}
```

Listing 8 SAXPY kernel vectorized using RISC-V Vector Intrinsics

```
void saxpy_rvv(size_t n,
               const float a,
               const float *x,
```

```
4                 float *y) {
5   for (int i = 0; i < n;) {
6     size_t vl = vsetvl_e32m8(n - i);
7     vfloat32m8_t vx, vy;
8     vx = vle32_v_f32m8(&x[i], vl);
9     vy = vle32_v_f32m8(&y[i], vl);
10    vy = vfmacc_vf_f32m8(vy, a, vx, vl);
11    vse32_v_f32m8(&y[i], vy, vl);
12    i += vl;
13  }
14 }
```

SIMD Loops

OpenMP SIMD, available as of version 4.0, provides a mechanism for semi-automated vectorization. OpenMP pragmas act as annotations with the added benefit that they are portable between compilers and architectures.

The main construct that enables SIMD support in OpenMP is the `simd` directive. This directive must immediately go before a loop and states that the iterations of the loop can be concurrently executed using SIMD instructions. A SIMD loop is executed by grouping the iterations of the original loop into *chunks* of consecutive iterations. The iterations in a single chunk are executed concurrently via SIMD instructions.

When using the `simd` directive, the compiler does not have to check about the correctness of the vectorized loop. This is an important distinction with some approaches that use annotations: in those the compiler adds to the facts, it could prove the facts provided by the user. In OpenMP the compiler assumes the programmer has assessed the validity of concurrent execution using SIMD instructions. Under these circumstances a compiler can skip a large part of the correctness analysis. However, there are still a number of decisions to be taken when vectorizing the code. OpenMP SIMD gives a lot of flexibility and control to the user to describe many facts about the loop being vectorized. This is done via a number of clauses.

- `simdlen`. The user can specify the preferred size of the chunk. This roughly corresponds to the vector length. Some architectures, mostly due to historical reasons, provide different vector lengths, and it may be necessary to specify the precise length due to their memory access patterns. Some loops may have slightly better performance when the chunk is reduced.
- `safelen`. Section "Vectorization, Intrinsics, and Semi-automatic Vectorization" has mentioned that loop-carried dependences must fit a vector. In OpenMP SIMD terms, this means that an iteration of a chunk cannot generate a value that is used by other iterations of the same chunk. A simple example is a loop that does `a[i+5] = a[i] + 1;`. If we use a vector longer than 5 elements, a chunk can be concurrently doing `a[5] = a[0] + 1; a[10] = a[5] + 1;` which gives unpredictable semantics to the value of `a[5]`.

How values evolve in a loop allows the compiler to emit more efficient code for the purpose of vectorization.

- `linear`. The value of a linear variable is described by a linear function with respect to the current iteration number. The scaling factor of the linear function is called *linear step* and by default is 1. `linear` is mainly useful for pointers or indexes of arrays incremented inside the loop. In absence of this information, the compiler should use a generic vector memory access like a *scatter* or *gather*. The linear clause allows the compiler to access memory using specialized vector memory accesses like unit stride or strided vector memory loads and stores.

Listing 9 `simd` directive applied to the loops in Listing 6

```
1  // Loop with no loop-carried data dependences. Any vector length is useable.
2  double y[], x[], a;
3  #pragma omp simd linear(w:2) nontemporal(y, x) uniform(a)
4  for (int i = 0; i < N; i++) {
5    int w = 2*i + 3;
6    y[i] += a*x[i] + w;
7  }
8  // Loop with loop-carried data dependences but still vectorizable
9  // with vectors with a length no larger than 5 elements.
10 double a[];
11 #pragma omp simd safelen(5) aligned(a)
12 for (int i = 0; i < N - 5; i++) {
13   a[i] = a[i] + a[i + 5];
14 }
```

Sometimes architectures impose constraints on the memory alignment of the data that can be operated using vector instructions. If the compiler cannot prove the alignment is suitable, it may have to use inefficient memory accesses when loading the vector from memory. The clause `align` allows stating the alignment of data.

Some algorithms expose low temporal locality, i.e., they do not reuse data that was previously accessed. Vector memory accesses need to bring from memory all the elements of a vector (or take to memory). This causes higher pressure to the cache system which leads to larger number of data being evicted. However, if the temporal locality is low, it may not be useful to cache the memory accesses. OpenMP SIMD clause `nontemporal` allows specifying that a variable exposes none to little temporal locality and memory accesses need not be cached.

Similar to OpenMP worksharings, described in section "The Worksharing Model", OpenMP SIMD loops can efficiently implement reduction patterns. In principle, reductions cannot be vectorized because their accumulating nature means they depend immediately on the previous iteration. However architectures provide specialized vector instructions that can perform an accumulation (e.g., an addition) through the elements of a vector. OpenMP SIMD allows to specify that a variable takes part in a reduction using the `reduction` clause. In that case the compiler uses a temporary vector initialized to the neutral value of the operation (e.g., 0 for an addition, 1 for a multiplication) where it performs the partial accumulations. At the end of the loop, the temporal vector is reduced using one of the specialized operations.

Listing 9 shows the loops in Listing 6 explicitly annotated with the `simd` directive. The directives inform the compiler that it is safe to vectorize the loops.

Listing 10 Example of loop using reduction

```
1 double a[];
2 double sum = 0.0;
3 #pragma omp simd reduction(+:s) nontemporal(a)
4 for (int i = 0; i < N; i++) {
5   s += a[i];
6 }
```

Function Vectorization

OpenMP SIMD imposes a number of constraints to a `simd` loop, for instance, not all the OpenMP constructs can be used in the body of the loop. However, calls to other functions in the loop body are allowed. This opens a number of possibilities when vectorizing code that makes function calls.

Traditionally, a loop that needs to call a function had the code of the invoked function inlined. This frees the vectorization algorithm of having special support for function calls. However it may not always be possible to inline functions, especially if they come from optimized libraries.

OpenMP SIMD allows the creation of SIMD versions of existing functions so they can be called from the body of a `simd` loop. This can be done using the `declare simd` directive. Calling a SIMD function from a SIMD loop means in general the scalar types of the parameter and return of the function are replaced by vector values.

Similar to the `simd` directive, OpenMP SIMD provides a lot of control regarding the SIMD function via clauses. These clauses are applied to the parameters of the function.

- `uniform`. The value of a parameter may be invariant during the execution of a loop. This is beneficial because parameter passing can pass a scalar rather than a vector. Also some usages of the parameter can be compiled to simpler operations: memory accesses using a uniform parameter can be done using scalars.
- `linear`. This has similar semantics to the `linear` clause used in a SIMD loop, but it allows specifying the modifiers `val`, `ref`, and `uval`. `val` has the same semantics as in a SIMD loop: the variable is a linear function of the iteration number. `ref` and `uval` are only useful for parameters that are passed by reference (as in C++ or Fortran). `ref` informs the compiler that the addresses of the passed references are linear (in other words, they are references in an array indexed by the iteration number scaled by *linear step*). `uval` informs the compiler that all the iterations use the same storage for the reference; thus, the referenced value will be `uniform`.
- `inbranch`/`notinbranch`. When vectorizing a loop, control flow (like another loop or if-then-else constructs) can be vectorized using a form of *if conversion* in which every element of the chunk is assigned a Boolean value. This

is often represented in a compact form called a *vector mask* or *vector predicate*. Vector masks allow the chunks to flatten the control flow by effectively enabling or disabling some of the elements of the chunk. When calling a function, it is important to know if the call happens inside control flow or not, so the vector version of the function needs to account for the vector mask. Not all architectures have good support for vectorizing control flow, and in such cases, it can be costly to vectorize those loops or functions. Other architectures have good support but still benefit from assuming they can operate on the whole vector, as this reduces the number of operands used by the vector instructions.

Listing 11 shows an example of `declare simd` that can be used to generate different SIMD versions of functions called from `simd` constructs.

Listing 11 Examples of `declare simd` to generate SIMD versions of functions

```
#include <math.h>

#pragma omp declare simd notinbranch
double exp(double x);

double y[], x[], a;
#pragma omp simd nontemporal(y, x)
for (int i = 0; i < N; i++) {
  y[i] += exp(-x[i]);
}

#pragma omp declare simd uniform(maximum, minimum)
double normalize(double x, double maximum, double minimum) {
  return 2 * ((x - minimum) / (maximum - minimum)) - 1;
}

double maximum = -DBL_MAX;
double maximum = DBL_MAX;
#pragma omp simd reduction(max: maximum) reduction(min: minimum)
for (int i = 0; i < N; i++) {
  maximum = y[i] > maximum ? y[i] : maximum;
  minimum = y[i] < minimum ? y[i] : minimum;
}

#pragma omp simd
for (int i = 0; i < N; i++) {
  y[i] = normalize(y[i], maximum, minimum);
}
```

The Accelerator Model

With the advent of heterogeneous platforms, mainly consisting of SMP cores and Graphics Processing Units (GPUs), and the possibility to use the latter for computing, OpenMP had the need to improve programmability on these environments. OpenMP selected to incorporate a new directive, the *target* extension to drive the code generation towards accelerators.

There are two main uses of the *target* directive: (i) provide a hint on which code regions should be translated for the accelerators and executed there [if possible] and (ii) provide a hint on extended code regions which must have a collection of data placed in the accelerator memory. Listings 12 and 13 show these two main uses.

Listing 12 Samples of OpenMP target directive on code

```
1  #pragma omp target parallel for
2  for (i=0; i<SIZE; i++)
3        ...
4
5  #pragma omp target teams
6  {
7      // code region
8  }
9
10 #pragma omp target teams distribute
11 for (i=0; i < SIZE; i++)
12       ...
```

Listing 13 OpenMP target data region

```
1  #pragma omp target data
2  {
3   // host code
4  #pragma omp target
5      // accel. code region 1
6   // host code
7    ...
8  #pragma omp target
9      // accel. code region 2
10 }
```

When working with accelerators, the execution environment must ensure that data is in the address space of the target accelerator. OpenMP achieves this goal by allowing the expression of the variables that need to be moved in and out of the accelerator memory: the *map* clause.

The *map* clause is used in both *target* and *target data* directives. It accepts a qualified list of symbols. For each one, data transfers are implemented following the qualifiers:

- *from*: data is moved to the accelerator prior to the execution of the code region.
- *to*: data is moved from the accelerator back to the host memory space, after the execution of the code region.
- *tofrom*: data is moved before the execution of the task to the accelerator and back after the execution.

Additional qualifiers (*alloc*, *release*, *delete*) allow to manage the mapping of the data directly on the accelerator.

Listing 14 shows the list traversal program, annotated to spawn work on an accelerator. The accelerated work is the initialization (lines 8–9) and the computation (lines 17–18) associated with each list element. The *target* directive behaves as a task, and it can also be provided with the data directionality hints, to implement a dataflow graph.

The OmpSs-2 Programming Model

OmpSs-2 is a data-flow programming model that extends and refines the original tasking model proposed in OmpSs (Duran et al. 2011) to cope with the increase in size, complexity, and heterogeneity of future Exascale systems. To that end, OmpSs-2 provides several advanced features, including a powerful dependency system that supports dependencies across different nesting levels and advanced dependency types such as *concurrent*, *commutative*, and task reductions; support to exploit large many-core processors, as well as NUMA systems. It is worth noting that all these features, implemented in the Mercurium compiler and Nanos6 runtime system, have been designed to be freely combined to ease the development of complex applications.

Listing 14 Example of use of the target directive

```
#pragma omp parallel
{
#pragma omp single
  {
    list_t list = begin;

    while (list != NULL) {
#pragma omp target depend(out: list) map(from: list->elem)
      initialize(list->elem);

      list = list->next;
    }

    list = begin;

    while (list != NULL) {
#pragma omp target depend(inout: list) map(tofrom: list->elem)
      compute(list->elem);

      list = list->next;
    }
  }
}
```

Advanced Dependency System

One of the challenges of parallel programming is the coordination of the joint activity of several cooperating activities. OpenMP (OpenMP Architecture Review Board 2020) and Cilk++ (Blumofe et al. 1995) support the fork-join model, which is especially well suited to exploiting structured parallelism. In the fork-join model, several independent tasks are spawned and executed until they reach an implicit synchronization point at the end of the fork-join construct. However, several studies have identified limitations on the fork-join execution model (Kurzak et al. 2010), especially for exploiting dynamic and irregular parallelism.

OmpSs pioneered the use of a tasking model for exploiting irregular parallelism, which was later adopted in OpenMP 3.0. In this model, independent tasks are

created and executed in parallel, until a *taskwait* – an explicit synchronization point – is reached. This tasking model is more flexible and suitable than the fork-join model to exploit irregular and nested parallelism. However, it lacks a mechanism for fine-grained synchronization. To overcome this issue, a data-flow model based on task dependencies was proposed by Duran et al. (2008) and adopted in OpenMP 4.0. To support it in the language, the *depend* clause was added to the task construct. From the point of view of its semantics, each task defines an inner, unique, and independent domain on which to calculate the dependencies of its direct children. OmpSs-2 extends this model by providing a global domain to calculate dependencies, as well as advanced dependency types such as *concurrent*, *commutative*, and *reduction*. Listing 15 shows how the reduction dependency type is used to parallelize the dot product of two vectors. On line 6, the *acc* variable is declared as a reduction dependency. This allows the parallel execution of all tasks instantiated inside the loop on line 4, each using a private copy of *acc* variable. The runtime system will automatically combine all the private copies of *acc* to calculate its final value, which will be consumed by the task defined on line 12.

Listing 15 Example of reduction dependency type

```
1  void dot_product(long N, long CHUNK_SIZE, double A[N], double B[N])
2  {
3      double acc = 0.0;
4      for (long i = 0; i < N; i += CHUNK_SIZE) {
5          long size = (N - i >= CHUNK_SIZE)? CHUNK_SIZE : (N - i);
6          #pragma oss task in(A[i;size], B[i;size]) reduction(+: acc)
7          {
8              for (long ii = 0; ii < size; ii++)
9                  acc += A[i + ii]*B[i + ii];
10         }
11     }
12     #pragma oss task in(acc)
13     printf("acc: %\n", acc);
14
15     #pragma oss taskwait
16 }
```

Global Domain of Dependencies

Since version 4.0, the tasking model of OpenMP supports both nesting and the definition of dependencies between sibling tasks. A natural way to parallelize many codes with tasks is to first taskify the high-level functions and then to further refine these tasks with additional subtasks. However, this top-down approach has some drawbacks since combining nesting with dependencies requires additional measures to enforce the correct coordination of dependencies across nesting levels. For instance, most non-leaf tasks need to include a *taskwait* at the end of their code. While these measures enforce the correct order of execution, as a side effect, they also limit the discovery of parallelism. In OmpSs-2 the OpenMP tasking model has been extended to improve the integration of nesting and dependencies (Perez et al. 2017). The design builds on both formulas, nesting and dependencies, and benefits from their individual strengths. On one hand, it encourages a top-down approach to parallelizing codes that also enables the parallel instantiation of tasks. On the other

hand, it allows the runtime to control dependencies at a fine grain that until now was only possible using a single domain of dependencies.

The combination of nesting and dependencies has three aspects that can be improved:

- The presence of the taskwait directive delays the completion of the enclosing task and thus the release of the system or user-level thread and its stack.
- A task with subtasks cannot release incrementally its own dependencies and those of its subtasks. Instead they are all released together once the task and all of its subtasks have finished.
- The presence of elements in the depend clause that are only needed by subtasks defers task start even when only subtasks need to be deferred.

Early Release of Dependencies
Detaching the taskwait at the end of a task from the task code allows the runtime to be made aware earlier of the fact that the task code has finished and that it will not create further subtasks. In a scenario with task nesting and dependencies, this knowledge allows it to make assumptions about dependencies. Since the task code is finished, the task will no longer perform by itself any action that may require the enforcement of its dependencies. Only the dependencies needed by its live subtasks need to be preserved. In most cases, these dependencies are the ones associated to an element of the depend clause that also appears in a live subtask. Therefore, the dependencies that do not need to be enforced anymore could be released. Moreover, this is equivalent to merging its inner dependency domain into that of its parent.

Weak Dependencies
On a parent task, each element of the *depend* clause may be needed only by the task itself, only by its subtasks, or by both. The elements that are only needed for the subtasks only serve as a mechanism to link the outer domain of dependencies to the inner one. In this sense, allowing these elements to defer the execution of the task is unnecessary, since the task does not actually perform any conflicting accesses by itself.

For this reason it has been proposed to extend the depend clause with three additional dependency types: *weakin*, *weakout*, and *weakinout*. Their semantics are analogous to the ones without the weak prefix. However, the weak variants indicate that the task does not perform by itself any action that requires the enforcement of the dependency. Instead those actions might only be performed by any of its nested subtasks. Any subtask that might directly perform those actions needs to include the element in its depend clause in the non-weak variant. In turn, if the subtask delegates the action to a subtask, the element must appear in its depend clause using at least the weak variant. Weak variants do not imply a direct dependency and thus do not defer the execution of tasks. Their purpose is to serve as linking point between the dependency domains of each nesting level. Weak dependencies, combined with the fine-grained release of dependencies, merge the inner dependency domain of a task into that of its parent. Since this happens at every nesting level, the result is

equivalent to an execution in which all tasks had been created in a single dependency domain. For more details please refer to Perez et al. (2017).

Advanced Dependency Types

Besides the standard *in*, *out*, *inout*, and weak variants mentioned above, OmpSs-2 also supports the following dependency types, which are a relaxed form of the *inout* dependency.

- concurrent: The *concurrent* dependence type behaves as the *inout* type with respect to *in*, *out*, and *inout* types, but has the particularity that no dependencies are enforced over other sibling tasks that define a concurrent type over the same memory reference. This dependency type has been introduced in OpenMP 5.1 as *inoutset*
- commutative: The *commutative* dependence type behaves as the *inout* type with respect to *in*, *out*, and *inout* types. It also enforces a dependence over other sibling tasks that define a commutative type over the same memory reference, but this dependence allows any order of execution between those tasks (as opposed to creation order). Any permutation ordering of those tasks annotated with commutative is correct, as long as only one of those tasks is executed at a time. This dependency type has been introduced in OpenMP 5.0 as *mutex_inoutset*
- reduction: As far as the interaction between dependence types is concerned, the *reduction* type behaves just as the *concurrent* type. The difference between them is that the last task annotated with a reduction clause will also be responsible for computing a reduction, but this has no implications from the point of view of the dependence model.

It is worth mentioning that reductions are also supported with array types (Pallares et al. 2019). To that end, the Nanos6 runtime system provides all the logic required to transparently manage private data copies and dynamically reduce them when they are required. This provides a seamless integration of reductions with other dependency types, without requiring any explicit scoping or global synchronization like in OpenMP reductions. Listing 16 shows how array reductions can be used to calculate the histogram of a set of images. In this example, the task loop construct is used to parallelize with tasks the inner loop, which computes the histogram of one single image. However, the use of array reductions enables the runtime system to process images belonging to different iterations in parallel.

Listing 16 Example of an array reduction

```
1 int img[N][size]; // init set of images ...
2 for (int iter = 0; iter < N; iter++){
3     #pragma oss taskloop in(img[iter][0;size]) \
4                 reduction(+: histo[0;size]) shared(img)
5     for (int i = 0; i < size; ++i) {
6         ++histo[img[iter][i]];
7     }
8 }
9 #pragma oss taskwait
```

Exploiting Structured Parallelism on Many-Core Processors

Shared memory programming models such as OpenMP provide work-sharing and task constructs. The former relies on the efficient fork-join execution model to exploit structured parallelism, while the latter relies on fine-grained synchronization among tasks and a flexible data-flow execution model to exploit dynamic, irregular, and nested parallelism. On applications that show both structured and unstructured parallelism, both work-sharing and task constructs can be combined. However, it is difficult to mix both execution models without penalizing the data-flow execution model. Hence, on many applications structured parallelism is also exploited using tasks to leverage the full benefits of a pure data-flow execution model. However, task creation and management might introduce a non-negligible overhead that prevents the efficient exploitation of fine-grained structured parallelism, especially on many-core processors. This chapter describes the OmpSs-2 *work-sharing* tasks proposed in Maroñas et al. (2019, 2020) and Maroñas (2021). These tasks internally leverage work-sharing techniques to exploit fine-grained structured loop-based parallelism.

The flexibility of the data-flow execution model relies on the dynamic management of data dependencies among tasks. However, dependency management might introduce a non-negligible overhead depending on the granularity and number of tasks. Hence, finding the adequate task granularity becomes a key point to get good performance: too many fine-grained tasks will increase task overheads, but too few coarse-grained tasks will hinder the available parallelism. Yet, it is not always possible to reach the optimal granularity that is coarse enough to compensate for the overheads while opening sufficient parallelism. Moreover, the granularity is limited by the problem size per core. Thus, if the problem size per core is too small, the granularity might be suboptimal, hurting the performance. For those situations, it makes sense to combine both strategies – tasking and work-sharing – in a way that drawbacks of each strategy can be mitigated while maximizing their strengths. To do so, an extension to tasks is proposed especially adapted to avoid the fork-join execution model.

Optimal Task Granularity

An inherent problem of task programming is granularity choice. Hence, if it is not adequately set, overhead may penalize overall performance. The overhead of tasks is caused by several different sources. The first one is the actual task creation, which usually implies costly dynamic memory allocations. The second one is the computation of dependencies between tasks, which involves the use of dynamic and irregular data structures. Finally, the scheduling of tasks across many cores can also become a bottleneck. Task granularity and the number of created tasks are inversely proportional. Consequently, a given problem can be solved either by using many fine-grained tasks or a few coarse-grained ones. Thus, finding an adequate granularity is a key point to optimally exploit resources when using tasks, alleviating the aforementioned overheads, but still creating enough parallelism to maximize resource utilization.

There are many situations where the problem size per core is not optimal, jeopardizing the benefits of using tasks:

- Strong scaling in distributed environments: This is a common scenario in HPC environments. Strong scaling starts from a given problem size per core and makes it smaller either by augmenting the number of resources or by decreasing the total problem size. As previously explained, reducing the problem size per core while maintaining the granularity of the tasks can lead to insufficient work.
- Many-core architectures: Increasingly, architectures have more and more cores. This trend directly affects the problem size per core, which becomes smaller because the same problem size is divided among more cores. Thus, setting an optimal granularity becomes harder or even impossible, because either tasks become too fine grained or there are not enough of them.
- Applications requiring different granularities: Many applications rely on different kernels to perform a computation, and each of them may require a different task granularity to achieve optimal performance. Finding an adequate granularity that fits all the different algorithms may be impossible because how data structures are data partitioned may implicitly fix task granularity. When this happens, it is especially important to have a wide set of granularities performing well in all the kernels.

Work-Sharing Task Syntax

A new clause for the task construct is proposed. This is the *for* clause for C/C++ and the *do* for Fortran. A *task for* – or worksharing task – accepts all the clauses accepted by a regular task except the final clause because a *task for* is always final. Note that being final means that no additional tasks will be created inside the context of a worksharing task. Additionally, it accepts the *chunksize(integer-expr)* clause. The integer expression specified as a chunksize sets the minimum chunk of iterations that each worker is going to execute when it requests work to the worksharing task, except for the last chunk that might contain fewer iterations. If not set, the default value is Tasksize/NumberOfCollaborators, so that each collaborator has at least one chunk to run. The *for* clause can only be applied to a task that is immediately succeeded by a loop statement, as shown in Listing 17. In this example, several *task for* are instantiated by the loop in line 1. Each task loop instance will execute the loop in line 3 in parallel as described above.

Listing 17 Examples of oss task for

```
1 for(i=0; i<PS; i+=TS)
2   #pragma oss task for chunksize(cs) inout(i)
3   for(j=i; j<i+TS; j++)
4     do_work(j);
```

Semantics of Work-Sharing Tasks

Regular tasks are executed entirely by a single worker concurrently, while a *task for* may be executed collaboratively by several workers, as a worksharing construct.

A worksharing task is like a regular task in this sense, and the synchronization is done through data dependencies or explicit synchronization points. Note that the data dependencies of the worksharing tasks are released when the last chunk is finished by the thread that runs that last chunk. As a worksharing construct, the iteration space of the for-loop is partitioned in chunks of chunksize size. The key point is that these chunks do not have the usual overheads associated with a task – such as memory allocation and dependency management. To run a chunk, a thread only needs the boundaries of that chunk and the data environment, much like worksharing constructs. So, in summary, a worksharing task can be run in parallel by multiple threads, better amortizing the task management overheads. Moreover, worksharing tasks enlarge the set of granularities that deliver good performance. In scenarios where only a few tasks are created and if these are not enough to keep all the resources busy, the use of worksharing tasks mitigates the lack of parallelism.

OmpSs-2 NUMA Support

Nowadays, processors with multiple sockets or multiple chiplets are proliferating. These kind of processors usually expose a single shared address space. However, due to hardware restrictions, they adopt a Non-Uniform Memory Access (NUMA) approach, where each processor or chiplet has faster access to the local memory and slower to the remote memories. Reducing data motion is crucial to improve the overall performance in such systems. Hence, computations should run as close as possible to where data is located.

The OmpSs-2 support for NUMA systems (Maroñas et al. 2021; Maroñas 2021) is composed of three main elements: an allocation API, a global directory, and a NUMA-aware scheduling policy. The allocation API allows application developers to properly allocate memory using different partitioning policies. Then, using the information provided by the user to specify dependencies between tasks, combined with the information collected in a global directory when allocating data, the Nanos6 runtime system performs NUMA-aware work scheduling, complemented with distance-aware and load balance-aware stealing.

NUMA-Aware Allocation API

This API has been designed to be simple but still flexible enough to support most common use cases. The API has two main functions: one to allocate data and another one to free it. Moreover, there are several additional functions to manipulate *bitmasks*, which are used to identify the NUMA nodes that will be used to allocate data. A NUMA node usually corresponds to a socket, but modern processors based on chiplets might also contain several NUMA nodes.

nanos6_numa_alloc_block_interleave(size, bitmask, block_size): This method serves to allocate data. Users should replace their regular allocation function (e.g., malloc, mmap, new, etc.) by this one. It allocates *size* bytes, interleaving blocks of *block_size* among the NUMA nodes specified in *bitmask*. This very simple function

offers a lot of possibilities. For instance, Listing 18 shows (1) how to spread data among all the available NUMA nodes (lines 6–8), (2) how to allocate all the data in a single NUMA node (lines 10–15), and (3) how to distribute data using a block-cyclic policy among all the available NUMA nodes (lines 17–20).

nanos6_numa_free(ptr): Users can invoke this method to release memory. *ptr* is the pointer to free, and it must be a pointer returned by the nanos6 NUMA allocation function. Apart from the memory release, it also removes the associated directory information.

Listing 18 Examples of nanos6 NUMA allocation method

```
1  nanos6_bitmask_t bitmask;
2  nanos6_bitmask_set_wildcard(&bitmask, NUMA_ALL);
3  int NUMA_nodes = nanos6_count_setbits(&bitmask);
4  size_t size = 4096*sizeof(int);
5
6  // Case 1: Distribute data among all the NUMA nodes
7  size_t block_size = size/NUMA_nodes;
8  void *A = nanos6_numa_alloc_block_interleave(size, &bitmask, block_size);
9
10 // Case 2: Allocate all data in a single NUMA node (node 0)
11 int node_id = 0;
12 nanos6_bitmask_clearall(&bitmask);
13 nanos6_bitmask_setbit(&bitmask, node_id);
14 size_t block_size = size;
15 void *B = nanos6_numa_alloc_block_interleave(size, &bitmask, block_size);
16
17 // Case 3: Block-cyclic among all the NUMA nodes
18 nanos6_bitmask_set_wildcard(&bitmask, NUMA_ALL);
19 size_t block_size = 512*sizeof(int);
20 void *C = nanos6_numa_alloc_block_interleave(size, &bitmask, block_size);
```

Nanos6 Data-Tracking System

To devise a proper work schedule, we need to know (1) where the data is, and (2) what data each task reads/writes. This section details how this knowledge will be obtained so it can be later used in the scheduling system. The data-tracking system contains a global directory that stores in which NUMA node resides the data allocated using the Nanos6 API. However, a single allocation may reside in several different NUMA nodes, since each of the pages can be placed in a different socket. The global directory is protected with a lock, to prevent data corruption. This could be a big source of overhead if not properly managed, because there could be many concurrent interactions with the directory, but only a single thread can access it at a time. Originally, a read-write lock was used. This kind of lock allows multiple readers to enter the protected area concurrently, while it only permits one single writer (and no readers) at a time.

Therefore, a further optimization was implemented to minimize the number of queries to the directory. During the dependency registration, when the newly created tasks are connected to predecessor tasks, the runtime iterates over the accesses of the task to check if there is an incomplete predecessor and the task must wait, or the access is ready to be used. During this step, the runtime also tries to propagate

the location of accesses. If either the parent or the predecessor of a created task has information about the location of an access, the runtime does not query the directory. In practice, this means that only the first task that accesses a data region performs a query, and all the rest are able to get it by propagation, reducing the number of queries up to a few orders of magnitude. Thanks to the previous optimizations, the overhead of the data-tracking mechanism is negligible when using adequate granularities.

Nanos6 NUMA-Aware Scheduling System

The scheduling system minimizes remote memory accesses by scheduling tasks to the NUMA node with better affinity. The NUMA-aware scheduling system contains one ready queue per NUMA node. To decide in which ready queue one has to add a task, an affinity score is computed. The runtime system enqueues the task in the NUMA node with the highest score. The runtime library computes the score of a task using the information of the data-tracking system detailed before. Each of the accesses of a task contains the location where it resides. Using the locations; the type of access (*in*, *out*, or *inout*); and the weights of each access, one can easily compute the amount of bytes that each NUMA node contains and derive a score. Each task contains its own accesses and can get the location, type, and weight of each access without any kind of synchronization, introducing a minimal overhead. Concretely, the score is the number of bytes of the task that a NUMA node contains.

So the tasks are enqueued in the ready queue of the NUMA node that obtained the highest score. Thus, when a compute unit requests work to the scheduler, the runtime system checks its NUMA socket and tries to get a task from the ready queue of that socket. If it is not possible, because there are no ready tasks, it steals tasks from the ready queues of other NUMA nodes. The stealing is done based on two factors: (1) the distance between the NUMA nodes and (2) the load balance. The Nanos6 runtime tries to steal tasks from closer NUMA nodes to reduce the NUMA effect penalty. However, if a NUMA node contains only a few tasks, the runtime steals from queues with a higher number of tasks to prevent further stealing, due to the associated NUMA effect penalty that it has.

Additionally, the runtime system implements an immediate successor mechanism. This mechanism tries to exploit temporal locality by skipping the regular logic of the scheduler and directly executing successor tasks. When a task finishes and releases its dependences, some of its successor tasks may become ready. If so, the finished task and the successor task are obviously sharing some data, so it is a good idea to run the successor task immediately in the same core to exploit the temporal locality. This mechanism has priority over the NUMA mechanism, because the temporal locality may expire while the spatial locality of the NUMA does not. Nevertheless, the runtime dynamically disables the immediate successor mechanism on a per-task basis considering the task data size. This is, if the runtime detects that the data set of a task is too big to fit in cache, and therefore cannot exploit temporal locality, it disables this mechanism to rely on the NUMA mechanism. Overall, this system minimizes data motion by moving compute to where data is rather than the other way around, mitigating the NUMA effect and improving performance.

Evaluation (Maroñas et al. 2021) has shown that the approach is able to outperform other state-of-the-art approaches such as the use of numactl across several different benchmarks in different platforms, being able to reach the optimal speedup in several of these benchmarks. Specifically, Nanos6 has been shown to obtain up to 2× speedup in a machine with two NUMA nodes and up to 7.1× speedup in a machine with 8 NUMA nodes, compared to the performance of running with a single NUMA node.

The XiTAO Programming Model and Runtime

XiTAO (The XiTAO development team 2021) is a lightweight layer built on top of modern C++ threading and concurrency features with the goals of being low-overhead and serving as a development platform for testing scheduling and resource management algorithms. XiTAO is built on a generalized task model that assembles (1) concurrency, (2) an embedded scheduler, and (3) a resizeable resource container. These TAOs (Task Assembly Objects) are moldable entities that can be scheduled into elastic resource partitioning, aka "elastic places." XiTAO targets better mapping between tasks and hardware resources such as cores, caches, or interconnect bandwidth. Therefore, among other features, XiTAO provides fast parallelism at low overhead, with constructive sharing and interference avoidance. Recently, XiTAO has been extended with novel high-level programming constructs and with introspective scheduling techniques that allow it to adapt to dynamic events as well as better target the goals of energy efficiency. Unlike OpenMP or OmpSs, XiTAO is built purely on top of standard C++ and does not require any additional compiler technology. Its goal is to prototype novel task scheduling technologies that can influence the standard and to explore methods for supporting heterogeneous programming in C++. One current direction is to extend current efforts in C++ and SYCL with ideas from XiTAO. Recently, `XiTAO` has also been extended to support pipeline parallelism via automatic tuning of pipeline stages and heterogeneous resources (Soomro et al. 2021).

Explicit DAG Programming in XiTAO

The `XiTAO` API has two levels. At the lower level is the explicit DAG interface, which allows to construct computational DAGs by creating new task assemblies (TAOs) and connecting them with edges. At the higher level lies the data-parallel interface which is described later.

`XiTAO` provides low-level constructs to describe TAO objects as C++ classes and interconnect them into a TAO-DAG. Listing 19 shows simplified structures of the base `AssemblyTask` class, a minimal TAO class, and the basic `XiTAO` API. TAOs are specified as derived classes from the base `AssemblyTask` class. The `AssemblyTask` needs to be initialized with a resource width and, optionally, an address in the software topology, which in the shown example (bottom of Listing 19)

corresponds to a 1D torus topology. TAOs can be created and pushed into the WSQs both before and after execution starts (via `xitao_start()`). Dynamic generation of TAOs is critical to support irregular TAO-DAGs. An example dot product code in XiTAO and the corresponding DAG are shown, respectively, in Listing 20 and Fig. 2.

Listing 19 TAO definition and `XiTAO` API

```
class AssemblyTask{ // defined by XiTAO
        int width;  // resources := #cores
        int leader; // id of first worker in team
public:
        int set_sta(float X);  // specify a STA (for a 1D topology)
        virtual int execute(int thread) = 0;
        int make_edge(AssemblyTask *); // create an edge
};

class TAO : public AssemblyTask { // defined by user or library
        TAO(args) : AssemblyTask(w){  // specify resource
                initialize(args);}        // initialize TAO
        int cleanup(){...};               // for garbage collection
        int execute(int thread){   // invoke inner scheduler
           int tid=thread-leader;  // obtain virtual id
           do_work(tid);}          // invoke scheduler and start computation
};

// XiTAO runtime management API
int xitao_init(int numthreads=all); // initialize the runtime
int xitao_start();  // start a XiTAO computation
int xitao_wait();   // wait for a computation to complete
int xitao_push(AssemblyTask *); // push a ready TAO into a WSQ based on the VTL

// XiTAO TAO-DAG API
TAO *obj1 = new TAO(w,...); // crate a new TAO with resource width 'w'
                            // TAOs are recycled by XiTAO after execution
obj1->make_edge(obj2); // add a dependency: obj2 depends on obj1
obj1->set_sta(0.5);    // set a 1D STA (=0.5)
obj1->clone_sta(obj2); // clone the STA of a TAO
```

Listing 20 `XiTAO` example showcasing the implementation of a dot product

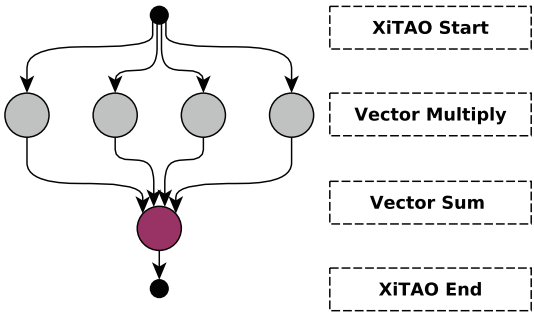
```
// ./a.out <veclength> <TAOwidth> <blocklength>
int
main(int argc, char *argv[])
{
  // C = A dot B, vector size is in 'len'
  double *A, *B, *C;
  // the result of the dot product
  double D;
  initialize_vectors(&A, &B, &C);

  xitao_init();
  // decomposition
  int numvm = len / block;

  // VecMul TAO implements vector
  //multiplication
  VecMul *vm[numvm];
  // VecAdd TAO implements
  vector reduction (+)
  // 'width' vector
  elements per TAO
  VecAdd *va = new VecAdd(C, &D, len, width);
```

```
23
24    // Create the TAO-DAG
25    for(int j = 0; j < numvm; j++){
26      vm[j] = new VecMul(A+j*block, B+j*block,
27      C+j*block, block, width);
28      //Create an edge
29      vm[j]->make_edge(va);
30      //Push current root to queue
31      xitao_push(vm[j]);
32    }
33
34    //Start the TAO-DAG execution
35    xitao_start();
36    //Finalize and claim resources back
37    xitao_fini();
38  }
```

Fig. 2 The dot product DAG



The initial TAO resource widths (i.e., the number of worker threads to execute the TAO) can be set explicitly or automatically using the internal performance modeling which is activated by adding `--perfmodel (-p)` to the `XiTAO` configuration options. `XiTAO` configuration parameters are described in section "Configuring the Runtime".

Software Topologies and Locality-Aware Programming

The Software Topology Mapping

XiTAO implements software topologies to achieve locality-aware scheduling of tasks in a portable manner. Since the execution model of XiTAO DAGs targets performance portability, any information for locality-aware scheduling needs to be generic and interpretable at runtime. At the task level, XiTAO implements a concept called "virtual topology" which is converted at runtime into actual thread mappings to enforce locality-aware scheduling. It is exposed using the `set_sta(float)` function available for each TAO. The function accepts a floating point number from 0.0 to 1.0, namely, the Software Topology Address (STA). The number represents a ratio that is to be interpreted by the runtime system to map the task to a hardware location. If two tasks have the same address, this is understood by the runtime as meaning the highest amount of data reuse between the two tasks. As a consequence,

the XiTAO runtime will attempt to schedule the two tasks on the same set of cores. This then optimistically results in data reuse via the caches of the cores. XiTAO currently implements a one-dimensional virtual topologies that is tested in one of its publicly available benchmarks (Heat).

Locality-Aware Moldable Mapping

In the case of single-threaded execution, the thread id that is mapped by the STA is the logical id of the smallest unit of execution exposed by the runtime environment, which is typically a hardware thread. However, to exploit the potential of moldability, the system resources can be expressed as elastic partitions. Elasticity refers to the ability of the scheduler to match the assigned resources to the workload requirements. The system in Fig. 3 shows an example hardware configured with two places with $W = 4$, four places with $W = 2$, and eight places with $W = 1$. LR indicates the "leader" thread id, that is, the id of the smallest thread in the place with width W. A place encompasses the cores that share a resource (e.g., cache level, memory subsystem, network element). Thus, the relative location resembled by the line segments underneath Fig. 3 points to a moldable resource partition (initially, $W = 1$ is selected). To allow for flexible allocation of resources, the scheduler optionally accepts a layout description file at initialization phase. For example, the dual-socket four-core system shown in Fig. 3 can be described using the file whose content is highlighted in Table 1. To configure XiTAO with a certain layout description file, set the variable `XITAO_LAYOUT_PATH` to the path of the file.

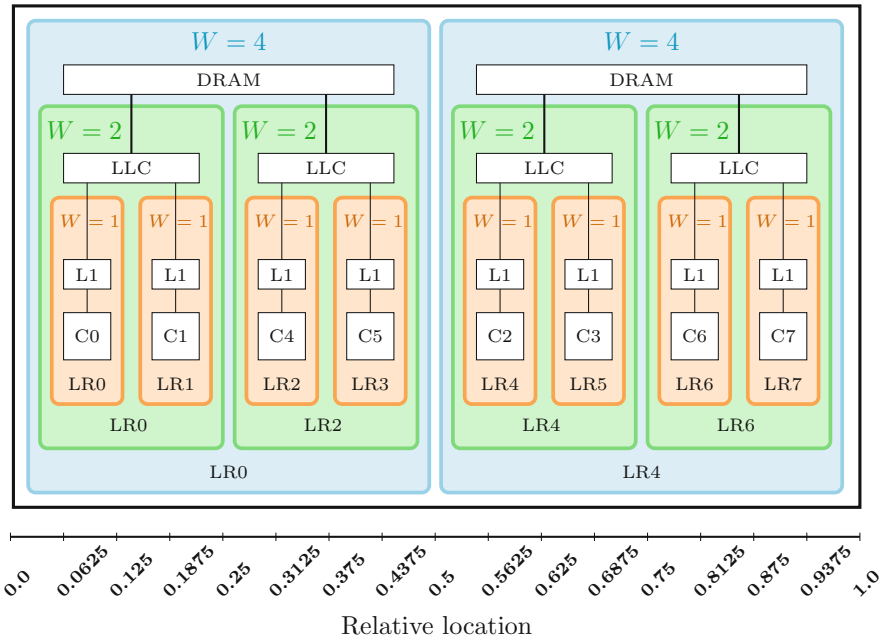


Fig. 3 An example mapping of relative STA location to the elastic resource partitions

Table 1 Example of the layout description

Line#	Content	Note
1	0,1,4,5,2,3,6,7	Thread affinity mappings
2	1,2,4	Widths for Leader thread 0
3	1	Widths for Leader thread 1
4	1,2	Widths for Leader thread 2
5	1	Widths for Leader thread 3
6	...	...

The XiTAO Data-Parallel Interface

This section describes enhancements provided on top of the XiTAO data-parallel interface. XiTAO incorporates modern C++ compiler technology to deliver a DAG-friendly data-parallel interface. With this interface, many applications that consist of parallel Single Program Multiple Data (SPMD) regions can leverage the backend features offered by the XiTAO RT including energy efficiency and interference awareness. The interface makes it possible to indicate a resource hint that the runtime can use to aggregate resource to a specific task within the loop or a set of tasks (resource moldability). Here, we make the distinction between the asynchronous and synchronous modes of executions. This is discussed in the following.

Listing 21 Basic structure of a DAG-based program inserting SPMD code regions (asynchronous mode)

```
// tao_width: XiTAO specific resource hint
// i: the loop counter
// loop_start: loop iterator start
// loop_end: loop iterator end
// scheduling_type: XiTAO scheduler type (e.g. dynamic)
// block_length: the chunk size for each task

auto dataparallel_nodes =
__xitao_async_data_parallel_region
  (tao_width, i, loop_start, loop_end,
   scheduling_type,  block_length,
     for (int j = 0; j < N; j++) {
       C[i][i] = 0;
       for (int k = 0; k < N; k++)
         C[i][j] += A[i][k] * B[k][j];
     }
);
// previous_node the parent node for the data parallel DAG
for(int i = 0; i < dataparallel_nodes.size(); ++i) {
  previous_node->make_edge(dataparallel_nodes[i]);
}
// attach the data parallel nodes to the next dependency
for(int i = 0; i < dataparallel_nodes.size(); ++i) {
  dataparallel_nodes[i]->make_edge(next_node);
}
```

The Asynchronous Data-Parallel Mode

This mode arises from the assumption that programs can be expressed as DAGs with different granularities. One of the main motivations behind the inclusion of async data-parallel nodes is the fact that task loops can then be seamlessly inserted into task graphs (see Fig. 4a) and will benefit from reducing the overhead of fork-join programming approaches and achieve energy efficiency from the runtime backend. The snippet on Listing 21 shows how a loop parallel region, for example, can be part of a full DAG structure using the XiTAO programming interface. Also, Table 2 highlights the interface parameters.

Listing 22 Basic structure of a DAG-based program inserting SPMD code regions (synchronous mode)

```
1 __xitao_sync_data_parallel_region (tao_width, i, loop_start,
2                                    loop_end, scheduling_type,
3                                    block_length,
4   for (int j = 0; j < N; j++) {
5     C[i][i] = 0;
6     for (int k = 0; k < N; k++) C[i][j] += A[i][k] * B[k][j];
7   }
8 );
```

The Synchronous Data-Parallel Mode

The sync data-parallel mode is semantically equivalent to OpenMP/OmpSs task loops and is mainly supported for backward compatibility of legacy codes. In this mode, operation happens in three steps, shown in Fig. 4b). First, the DAG execution

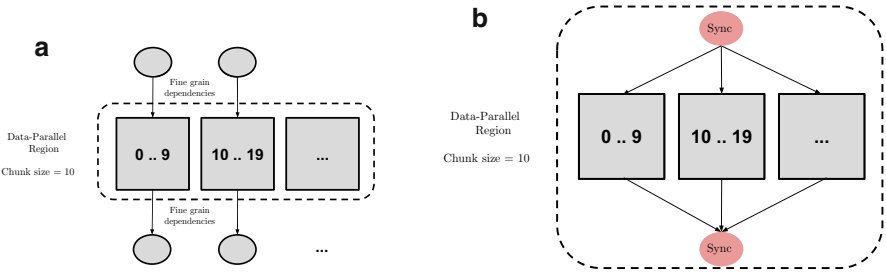


Fig. 4 XiTAO data-parallel modes. (**a**) Asynchronous mode with fine-grained dependencies. (**b**) Synchronous mode that is analogous to fork-join models

Table 2 The parameters input by user to the XiTAO's asynchronous data-parallel interface

Parameter	Usage
`width`	The XiTAO resource hint to be given to the loop tasks.
`iter`	The loop index/iterator.
`end`	The loop end.
`sched`	The scheduling options (e.g., static, dynamic, energy-aware, etc.)
`block_size`	Governs the granularity of task creation.

of previous nodes is synced. Second, the loop is divided into chunks of tasks according to the `block_length` parameter. Third, an implicit wait is inserted to pause the execution until all loop tasks have finished. Listing 22 shows an example of such usage.

The XiTAO Runtime

This section describes XiTAO runtime internals that are important for optimizations and to avoiding synchronization errors. A full description of the moldable task scheduler and queuing system is described in Pericàs (2018).

`XiTAO` Internals

An important design consideration in `XiTAO` is the queuing system and the implementation of moldability. Figure 5 depicts a high-level view of the queuing system. `XiTAO` maintains two queues for each core. TAOs first encounter the work stealing queues (WSQ). These queues are similar to other work stealing runtimes such as Cilk. The WSQs are used to balance the load across the cores. Note that the WSQs are agnostic to moldability, i.e., they manage the TAOs as if they were simple single-threaded tasks. Only upon selection of a ready TAO does the execution take moldability into account. To this end, `XiTAO` implements a second set of queues for each core, called Assembly Queues (AQ). The AQs are managed in a strict FIFO policy. Upon selecting a ready TAO, the runtime allocates a set of AQs corresponding to the subset of cores on which the TAO is to be executed in a worksharing manner and inserts a reference to the TAO into each of these

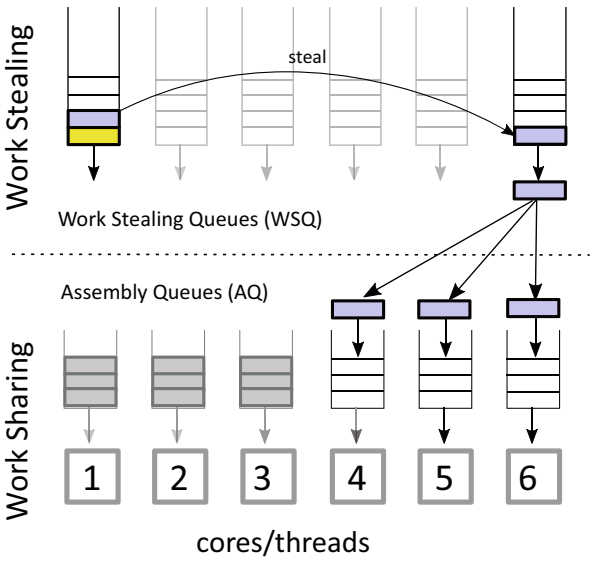


Fig. 5 The architecture of XiTAO's moldable task scheduler. This figure is a simplified version of the figure in Pericàs (2018)

queues. To avoid potential deadlock, locks to each AQ are taken before insertion takes place, in order to make this operation look atomic. The cores then execute the TAOs by extracting the TAO references from the AQ, one by one, and invoking its `execute()` method. More details on `XiTAO`'s queuing system can be found in Pericàs (2018).

This design has certain implications when programming TAOs. The first is that TAOs can only synchronize internally. They should never attempt to synchronize externally, as this would likely lead to a deadlock. This property makes TAOs look similar to CUDA thread blocks, or OpenCL work groups. The second implication is that the cost of inserting ready TAOs is proportional to the number of cores. Hence, narrow TAOs are preferable when the application is composed of very fine-grained tasks.

Execution of TAOs from the AQs does not imply a barrier at starting or finalization of each TAO. Hence, a large degree of asynchrony and overlapping is possible across different TAOs. However, whether this is beneficial depends very much on the characteristics of the application and whether the TAOs are compute-bound, cache-bound, or memory-bound. A `XiTAO` compile time option can be used to for barrier synchronization each time a TAO starts/finishes.

Configuring the Runtime

To configure the XiTAO runtime scheduler, there are certain features that can be enabled/disabled. This section briefly highlights such features. They are configured as command line options parsed from the string value (i.e., `--xitao_args="[args]"`). By passing the command line arguments to the runtime using `xitao_init(argc, argv)`, the DAG application can be run using `./bin --xitao_args="-h"` to get the available options as follows:

```
Usage: --xitao_args= [options]
Long option (short opt)   : Description (Default value)
 --wstealing (-w) [0/1]   : Enable/Disable work stealing (1)
 --perfmodel (-p) [0/1]   : Enable/Disable performance modeling (1)
 --nthreads (-t)          : The number of worker threads (8)
 --idletries (-i)         : The idle tries before a steal attempt
                            (100)
 --minparcost (-c) [0/1]  : Model 1 (parallel cost) - 0
                            (parallel time) (1)
 --oldtickweight (-o)     : Weight of old tick versus new tick (4)
 --refreshtablefreq (-t)  : How often to try a random width (10)
 --mold (-m)              : Enable/Disable dynamic moldability (1)
 --help (-h)              : Show this help
```

Conclusion

This chapter has provided an overview of the current practice for programming parallel computing systems, with a focus on programming models targeting a single node. Three models are introduced in detail: OpenMP, OmpSs, and XiTAO.

As energy efficiency becomes a strict requirement and systems become more heterogeneous, the programmability of accelerators is now a major concern. This is highlighted by describing techniques for exploiting SIMD instructions and for programming accelerators. Both these are discussed in the context of OpenMP. Finally, as systems become more hierarchical with multiple levels of caches and NUMA nodes, supporting NUMA-aware programming and automatically tuning resources become critical for achieving best performance. Features to achieve high performance in the NUMA context are described at the level of OmpSs and XiTAO.

Acknowledgments This work has been developed with the support of the Spanish Ministry of Science and Innovation (Computacion de Altas Prestaciones VIII: PID2019-107255GB). This work has received funding from the European Union Horizon 2020 research and innovation program under the LEGaTO project with grant agreement No. 780681 (https://legato-project.eu/). This work is supported by the European Union's Horizon 2020 research and innovation program under the grant agreement No 754304 (DEEP-EST). This work has been done as part of the European Processor Initiative project. The European Processor Initiative (EPI) (FPA: 800928) has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement EPI-SGA1: 826647

References

Blumofe RD, Joerg CF, Kuszmaul BC, Leiserson CE, Randall KH, Zhou Y (1995) Cilk: an efficient multithreaded runtime system. In: Proceedings of the fifth ACM SIGPLAN symposium on principles and practice of parallel programming, PPOPP'95, pp 207–216

Duran A, Perez JM, Ayguadé E, Badia RM, Labarta J (2008) Extending the openmp tasking model to allow dependent tasks. In: Eigenmann R, de Supinski BR (eds) OpenMP in a new era of parallelism. Springer, Berlin/Heidelberg, pp 111–122. ISBN 978-3-540-79561-2

Duran A, Ayguadé E, Badia RM, Labarta J, Martinell L, Martorell X, Planas J (2011) OmpSs: a proposal for programming heterogeneous multi-core architectures. Parallel Process Lett 21(2): 173–193

Karrenberg R (2015) Automatic SIMD vectorization of SSA-based control flow graphs. Springer, Wiesbaden

Kennedy K, Allen JR (2001) Optimizing compilers for modern architectures: a dependence-based approach. Morgan Kaufmann Publishers Inc., San Francisco

Kurzak J, Ltaief H, Dongarra J, Badia RM (2010) Scheduling dense linear algebra operations on multicore processors. Concurr Comput Pract Exp 22(1):15–44. ISSN 1532-0626

Larsen S, Amarasinghe S (2000) Exploiting superword level parallelism with multimedia instruction sets. In: Proceedings of the ACM SIGPLAN 2000 conference on programming language design and implementation, PLDI'00. Association for Computing Machinery, New York, pp 145–156. ISBN 1581131992. https://doi.org/10.1145/349299.349320

Maroñas M (2021) On the design and development of programming models for exascale systems. PhD thesis, Universitat Politècnica de Catalunya

Maroñas M, Sala K, Mateo S, Ayguadé E, Beltran V (2019) Worksharing tasks: an efficient way to exploit irregular and fine-grained loop parallelism. In: 2019 IEEE 26th international conference on high performance computing, data, and analytics (HiPC). IEEE, pp 383–394

Maroñas M, Teruel X, Bull M, Ayguade E, Beltran V (2020) Evaluating worksharing tasks on distributed environments. In: 2020 IEEE international conference on cluster computing (CLUSTER). IEEE. Pending publication

Maroñas M, Ayguadé E, Beltran V (2021) Mitigating the NUMA Effect on Task-Based Runtime Systems. Submitted to the Journal of Supercomputing. ACM

OpenMP Architecture Review Board (1997) OpenMP Fortran Application Programming Interface 1.0. Accessed: 18 Feb 2021

OpenMP Architecture Review Board (1998) OpenMP C and C++ Application Programming Interface 1.0. Accessed: 18 Feb 2021

OpenMP Architecture Review Board (2005) OpenMP Application Programming Interface 2.5. Accessed: 18 Feb 2021

OpenMP Architecture Review Board (2008) OpenMP Application Programming Interface 3.0. Accessed: 18 Feb 2021

OpenMP Architecture Review Board (2013) OpenMP Application Programming Interface 4.0. Accessed: 18 Feb 2021

OpenMP Architecture Review Board (2020) OpenMP Application Programming Interface 5.1. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf. Accessed: 18 Feb 2021

Pallares F, Mateo S, Beltran V, Ayguadé E (2019) Master Thesis: extending OmpSs-2 with flexible task-based array reductions. https://upcommons.upc.edu/handle/2117/129246. Accessed: 01 Mar 2021

Perez JM, Beltran V, Labarta J, Ayguadé E (2017) Improving the integration of task nesting and dependencies in OpenMP. In: 2017 IEEE international parallel and distributed processing symposium (IPDPS). IEEE, pp 809–818

Pericàs M (2018) Elastic places: an adaptive resource manager for scalable and portable performance. ACM Trans Archit Code Optim 15(2). ISSN 1544-3566. https://doi.org/10.1145/3185458

RISC-V Community (2021) RISC-V vector extension intrinsic document. https://github.com/riscv/rvv-intrinsic-doc. Accessed: 25 Feb 2021

Soomro PN, Abduljabbar M, Castrillon J, Pericás M (2021) An online guided tuning approach to run CNN pipelines on edge devices. In: Proceedings of the 18th ACM international conference on computing frontiers (CF'21), New York. Association for Computing Machinery (ACM) pp 45–53. ISBN 9781450384049. https://doi.org/10.1145/3457388.3458662

Swamy H (2012) Structured parallel programming patterns for efficient computation by michael mccool, arch d. robison and james reinders. ACM SIGSOFT Softw Eng Notes 37:43. https://doi.org/10.1145/2382756.2382773

The XiTAO development team (2021) XiTAO. https://github.com/CHART-Team/xitao.git. Accessed: 26 Feb 2021

Dataflow Models of Computation for Programming Heterogeneous Multicores

31

Jeronimo Castrillon, Karol Desnos, Andrés Goens, and Christian Menard

Contents

Introduction . . . 1108
About Models of Computation . . . 1110
Dataflow Models of Computation . . . 1112
Static Dataflow Models . . . 1112
Dynamic Dataflow MoCs . . . 1120
Reconfigurable Dataflow . . . 1122
Optimization of Dataflow Programs . . . 1125
Modeling Heterogeneous Platforms . . . 1126
Static Mapping . . . 1129
Hybrid Mapping . . . 1132
Examples: Models and Tools . . . 1134
Dataflow in Commercial and Mainstream Tools . . . 1134
MPSoC Application Programming Studio (MAPS) . . . 1134
PREESM and SPIDER . . . 1137
Conclusion and Outlook . . . 1140
References . . . 1140

The work was carried out by Andrés Goens while at the Chair for Compiler Construction, TU Dresden

J. Castrillon (✉) · C. Menard
Chair for Compiler Construction, TU Dresden, Dresden, Germany
e-mail: Jeronimo.Castrillon@tu-dresden.de; christian.menard@tu-dresden.de

K. Desnos
Univ Rennes, INSA Rennes, CNRS, IETR-UMR6164, Rennes, France
e-mail: karol.desnos@insa-rennes.fr; kdesnos@insa-rennes.fr

A. Goens
School of Informatics, The University of Edinburgh, Edinburgh, UK
e-mail: andres.goens@ed.ac.uk; andres.goens@tu-dresden.de

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_45

Abstract

The hardware complexity of modern integrated circuits keeps increasing at a steady pace. Heterogeneous Multi-Processor Systems-on-Chips (MPSoCs) integrate general-purpose processing elements, domain-specific processors, dedicated hardware accelerators, reconfigurable logic, as well as complex memory hierarchies and interconnect. While offering unprecedented computational power and energy efficiency, MPSoCs are notoriously difficult to program. This chapter presents MoCs as an appealing alternative to traditional programming methodologies to harness the full capacities of modern MPSoCs. By raising the level of abstraction, MoCs make it possible to specify complex systems with little knowledge of the target architecture. The properties of MoCs make it possible for tools to automatically generate efficient implementations for heterogeneous MPSoCs, relieving developers from time-consuming manual exploration. This chapter focuses on a specific MoC family called dataflow MoCs. Dataflow MoCs represent systems as graphs of computational entities and communication channels. This graph-based system specification enables intuitive description of parallelism and supports many analysis and optimization techniques for deriving safe and highly efficient implementations on MPSoCs.

Keywords

Heterogeneous multicores · Models of computation · Dataflow programming · Software abstraction · Streaming models · Compilers · Application mapping · Scheduling · Multicore simulation · Design space exploration

Introduction

The computer architecture landscape is vast, including application-specific accelerators and processors, complex interconnect and memory subsystems, as well as reconfigurable and emerging non-Von Neumann architectures. This hardware complexity is a result of the diminishing returns of technology scaling in combination with the ever-increasing demands from computing domains like big data, automotive, or Industry 4.0. Systems today integrate heterogeneous cores and heterogeneous memories and are interconnected to form complex computing systems, like Cyber-Physical Systems of Systems (CPSoSs). Evidence of this system complexity is provided in other sections and chapters of this book. While other chapters in this section have described tool flows to efficiently design such systems, this chapter focuses on sound programming methodologies to manage the system complexity.

The complexity of programming parallel, heterogeneous, and interconnected systems is widely acknowledged as one of the biggest challenges in computing. Achieving portable performance and improving debuggability and traceability are

just examples of open questions across computing domains. In the embedded domain, in particular, applications often come with nonfunctional requirements that further complicate software development. Examples are safety and mixed-criticality concerns, real-time constraints, predictable and deterministic behavior, energy efficiency for battery-powered devices, or robustness and reliability for harsh environments. There is a large body of research on these different aspects of software development, e.g., from the static code analysis community (Wilhelm et al. 2008), from the verification and model checking standpoint (Leroy 2009), from real-time systems community (Alur et al. 1990), and from the Electronic Design Automation (EDA) community to name a few. This chapter focuses on programming methodologies to deal with a subset of the requirements listed above.

Several parallel programming models and methodologies have been proposed since the advent of multicores in the mid-2000s. Large efforts have gone into extending and adapting general-purpose Application Programming Interfaces (APIs) and programming models for parallel computing to embedded platforms. Today, many embedded platforms support Pthreads, OpenMP, OpenCL, and alike, providing embedded programmers with rich environments and convenient interfaces known from general-purpose computing. However, initiatives for embedded systems like the Heterogeneous System Architecture (HSA) (Rogers and Fellow 2013) or the task and resource APIs from the Multi-Core Association (MCA) (Gleim and Levy 2013) better cater for the resource-constrained and highly heterogeneous nature of embedded systems. While of high practical value, most of these Application Programming Interfaces (APIs) do not provide enough *control* over the execution semantics of parallel applications. Directly programming with threads, for instance, is known to be hard in its own right (Lee 2006) and makes attempts to guarantee the nonfunctional requirements discussed above almost futile. This chapter advocates for a more principled approach to programming models, based on formal Models of Computation (MoCs), with an emphasis on the family of dataflow MoCs. Dataflow is a paradigm from the 1960s that was heavily researched from the 1980s with high success in the embedded domain. Dataflow models restrict the programmer, allowing more control for automated methodologies to analyze and optimize application implementations, providing a variety of guarantees on application requirements. The implementations themselves can leverage lower-level APIs for parallel computing.

This chapter introduces the general concept behind MoCs (section "About Models of Computation") and provides a review of the most prominent dataflow MoCs (section "Dataflow Models of Computation"), with insight into desirable properties for automated methodologies. Section "Optimization of Dataflow Programs" then describes general analysis and optimization flows, highlighting the commonalities found in the state of the art. To bring these general concepts closer to implementation, section "Examples: Models and Tools" further discusses commercial and academic tools. This includes details on two academic tools, namely, MAPS (Ceng et al. 2008) and PREESM (Pelcat et al. 2014), explaining what models underlie these tools and what methodologies they implement.

About Models of Computation

In general, a *model* is a mathematically grounded representation capturing predictable characteristics of a system. A model consists of a set of elements that can be assembled to describe a system, respecting a set of rules. For a valid representation built with a model, mathematical equations are usually associated with the elements of the model and make it possible to predict characteristics of the modeled system.

A *Model of Computation (MoC)* describes the behavior of a computing system, generally consisting of a combined data and control flow. The data flow of a system specifies how this system receives, moves, and produces data; and its control flow specifies when, if, and how the system processes this data. As presented in Jantsch (2003), MoCs can be seen as an interface between the computer science and the mathematical domains. A MoC specifies a set of rules that control how systems described with the MoC are executed. The term MoC is mostly used by the computer engineering and signal processing communities, while the computer science community uses the term *computing paradigm* for the same concept. A MoC is a set of operational elements that can be composed to describe the behavior of a system. The set of operational elements of a MoC and the set of relations that can be used to link these elements are called the *semantics* of this MoC.

Figure 1 shows an example and the semantics of the well-known Finite-State Machine (FSM) MoC. The semantics consists of two elements: a finite set of states and transitions. In an FSM, a single state is active at any given time and state changes are only possible via transitions, which are generally triggered by a condition on an input of the system. For example, the FSM of Fig. 1a models a controller for a simplified automatic transmission with three gears: `Park`, `Drive`, and `Reverse`. Starting from the initial `Park` state, the system can transition to the `Drive` or `Reverse` states depending on an input given by its user: move forward or move backward. The system will go back to the `Park` state only when the speed V of the system becomes zero.

The MoC is the interface between system developers and the mathematical domain, where characteristics of a system described with a MoC can be verified

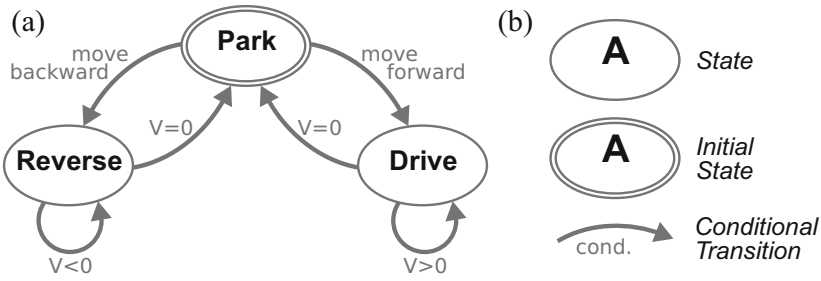


Fig. 1 Finite-State Machine (FSM) semantics and example. (**a**) Automatic transmission controller FSM. (**b**) FSM semantics

and proved independently of implementation concerns. Once a satisfactory system specification with a MoC is reached, an actual implementation can be built by translating the semantics of this MoC into a program construct, in any programming language supporting the execution rules of this MoC. For example, the FSM of Fig. 1a can be translated into the two implementations presented in Listings 1 and 2, in C and VHDL, respectively.

The concept of semantics in MoCs should not be mistaken with the concept of syntax of programming languages. The syntax of programming languages enables

Listing 1 C implementation of the FSM from Fig. 1a

```
enum State s = Park;
while(1){
  switch(s){
    case Park:
      if(move_forward)
        s = Drive;
      else
        if(move_backward)
          s = Reverse;
      break;
    case Drive:
      if(v == 0)
        s = Park;
      break;
    case Reverse:
      if(v == 0)
        s = Park;
      break;
  }
}
```

Listing 2 VHDL implementation of the FSM from Fig. 1a

```
signal state = Park
process(clk)
begin
  case state is
  when Park =>
    if move_forward = '1' then
      state <= Drive;
    elsif move_backward = '1' then
      state <= Reverse;
    end if;
  when Drive =>
    if v = 0 then
      state = Park;
    end if;
  when Reverse =>
    if v = 0 then
      state = Park;
    end if;
  end case;
end process;
```

the construction of programs that are executed using one or several underlying MoCs. For example, programs written in languages like Python or C are originally intended to be executed on processors, while programs written in the Verilog language are intended to be synthesized into logic circuits. The C or assembly languages adopt the imperative MoC, and C++ or Java adopt imperative, object-oriented, and a bit of functional MoCs. A language can also be used to implement a MoC not naturally implemented by its syntax, as illustrated in Fig. 1a.

The advantage of using a MoC to specify the behavior of a system is that in return for abiding by the MoC semantics, key properties of the system can be specified, guaranteed, and verified by construction. For example, in an FSM, all possible transitions between states of the system are explicitly specified, and thus no other transitions are possible. This is particularly useful to prevent unwanted or harmful behavior when specifying a system, assuming the actual implementation fully enforces the semantics of the MoC. The transmission controller in Fig. 1a, for example, does not allow any transition between the `Drive` and `Reverse` states, which would likely damage the controlled system. Guaranteeing such properties from a system specification with a MoC is generally much simpler than verifying them on the corresponding implementation with a programming language.

A plethora of MoCs have been created for diverse purposes. Some models, like FSMs, are great to specify the behavior of control-oriented systems but do not capture any concurrency and only very limited data processing. Other models, like lambda calculus (Church 1985), focus on the specification of the computational part of a system. This chapter focuses on a family of MoCs, namely, dataflow MoCs, which is largely used for the specification of systems processing streams of data, such as signals, videos, or images. The properties of the dataflow MoCs, detailed in next sections, make them particularly suitable for implementation on modern integrated circuit technologies. For more complete surveys of existing MoCs and their properties, refer to Lee and Seshia (2016).

Dataflow Models of Computation

Static Dataflow Models

Engineers, scholars, and researchers commonly use sketches to conceptualize and illustrate systems and ideas in early stages of development. These informal graphical representations are used in many engineering domains to specify the purpose of the components of a system and the relations between these components. Block diagrams are among the most popular informal models for the high-level specification of electronic and computing systems. A block diagram is a drawing composed of a set of boxes, usually representing the components of the system, and a set of lines and arrows representing the relations between components. In 1961, Kelly et al. introduced the BLOck DIagram compiler (BLODI), the first compiler for programs described with a formalized block diagram language. At the time of

punched cards, the main objective of the BLODI language was to make computer programming accessible to persons with no programming language knowledge. The BLODI language was composed of a set of 30 primitive blocks, like adders, filters, and quantizers, that could be used to compose Digital Signal Processing (DSP) applications. In 1974, Kahn (1974) and Dennis (1974) independently created the first mathematically grounded MoCs based on graphs, laying the foundation for the dataflow MoCs family.

A common semantics of all dataflow MoCs is the specification of systems with directed graphs called dataflow graphs or process networks. A few elements found in this common dataflow semantics are:

- **Actors**: The vertices of a dataflow graph, generally called actors or processes, represent the computational entities of the dataflow MoCs. Each actor of the graph can consume and produce data on a set of data ports and perform some processing using or generating this data.
- **Communication channels**: The edges of a dataflow graph represent communication channels used for transmitting data between actors, usually assuming a First-In, First-Out queue (FIFO) mechanism.
- **Data tokens**: The atomic pieces of data transiting on the edges of a dataflow graph are called data tokens. The type of data abstracted by a single data token depends on the specified system needs. A data token can be a simple 8-bit integer value, a complex number with two floating point values, or even a whole 2D image with millions of pixels. Heterogeneous data token types can coexist in a single dataflow graph, with each communication channel generally associated with a single type of data token.

The specification of the internal behavior of actors is not always an integral part of the MoC semantics. Instead, dataflow MoCs generally specify a set of rules which govern when actors are allowed to consume and produce a certain number of tokens (Lee and Messerschmitt 1987).

The popularity of dataflow MoCs for the design of stream processing systems notably comes from the assets they offer for deriving efficient implementations on modern hardware and software technologies. Implementing efficient software on modern hardware requires allocating processing, memory, communication, and energy resources for each part of the system. By clearly exposing separate computational entities, data movements, and computation triggers, dataflow MoCs ease this implementation process. Another key advantage of dataflow MoCs is their expressiveness for concurrent computations and data movement, which is an essential feature for both hardware design and parallel software design (Ecker et al. 2009). Finally, dataflow MoCs have been the topic of many research works, and many analysis and optimization techniques can be found in the literature, some of which are presented in section "Optimization of Dataflow Programs".

After this introduction to the basic concepts and advantages of dataflow MoCs, the following sections present the semantics of a few of the most popular models.

Homogeneous Synchronous Dataflow (HSDF)

The Homogeneous Synchronous Dataflow (HSDF) model, one of the simplest dataflow MoCs, was introduced by Lee and Messerschmidt in Lee and Messerschmitt (1987). The graphical semantics and an example of a HSDF graph are presented in Fig. 2.

Formally, an HSDF graph is noted $G = \langle A, F \rangle$, where A is the set of actors and $F \subseteq A \times A$ is the set of FIFOs of the graph. An actor $a \in A$ is defined as a function $a : F^n \rightarrow F^m$, consuming data tokens on $n \in \mathbb{N}$ input FIFOs and producing data tokens on $m \in \mathbb{N}$ output FIFOs.

In the HSDF MoC, each atomic execution of an actor, called a firing, consumes exactly one data token on each input port and produces exactly one data token on each output port. The HSDF execution semantics are entirely data driven, meaning that the availability of data tokens in each input FIFO of an actor is an enabler for its execution. This semantics eases the specification of task parallelism, as two actors with sufficient tokens in their input FIFOs can be fired concurrently. For example, in the HSDF graph from Fig. 2a, actor A can be fired anytime, since it has no input port. Each firing of actor A produces data tokens that can in turn trigger executions of actors C and D, possibly in parallel, and also of actor B if a data token is available on its second input port, connected to actor G. From the HSDF-MoC-theoretical point of view, there is no notion of time captured by the model. Consequently, an actor firing is modeled as an event instantaneously consuming and producing the tokens on inputs and output ports, respectively.

To ensure the liveness of cyclic data paths, that is, the absence of deadlocks when executing the HSDF graph, delays must be used to specify the presence of data tokens in a FIFO at initialization time. For example, in the HSDF graph of Fig. 2a, the delay in the FIFO (G, B) ensures the liveness of the cycle $B \rightarrow E \rightarrow G \rightarrow B$. A simple technique to verify the liveness of an HSDF graph is to check the acyclicity of the graph after removing all FIFOs with a delay. For a live HSDF graph, a graph iteration refers to executing each actor in the graph exactly once while respecting data dependency, thus bringing the graph back to its initial state.

HSDF actors are assumed to be stateless, which means that the values of data tokens produced during a firing of an actor solely depend on the values of data

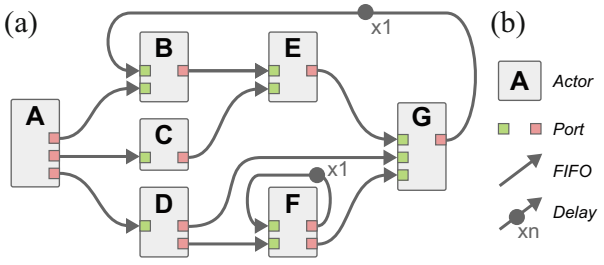


Fig. 2 Homogeneous Synchronous Dataflow (HSDF) semantics and example. (**a**) HSDF graph example. (**b**) HSDF graphical semantics

tokens consumed at this firing. The stateless property of HSDF actors enable the specification of *data parallelism*, that is, multiple firings of an actor can be triggered in parallel if enough data tokens are available on its input FIFOs. Data parallelism is a powerful feature of the HSDF MoC that enables specifying highly parallel computations with graphs containing only a few actors. If an HSDF actor must maintain an internal state containing useful data for future firings, this state must be explicitly modeled in the graph with a self-loop FIFO (see (F, F) in Fig. 2a). Such a self-loop FIFO forces the firings of the enclosed actor to happen sequentially, thus loosing all potential parallelism for this actor.

The HSDF MoC is particularly useful for designing stream processing applications running on Multi-Processor Systems-on-Chips (MPSoCs). For example, deriving an implementation of an HSDF graph on a multicore target with a shared memory requires solving two resource allocation problems: *mapping and scheduling* as well as *memory allocation*.

Mapping and scheduling: Mapping is the process of assigning firings of each HSDF actor to a processing element of a target architecture (Lee and Ha 1989). Scheduling is the process of determining the execution order of firings mapped to the same processing element. A common way to map and schedule an HSDF graph is to compute, at design time, a mapping and schedule for a complete graph iteration. This pre-computed per-processor schedule can then be repeated indefinitely in order to execute the application. A simple mapping and scheduling heuristic consists in traversing the HSDF graph in topological order, ignoring FIFOs with delays, and assigning each actor to the first processing element available.

The Gantt diagrams in Fig. 3 illustrate possible schedules of the HSDF graph from Fig. 2a on an architecture with two cores. In Fig. 3a, graph iterations do not overlap, and the execution of the last firing of actor G must end before a new iteration begins with a new execution of actor A. In Fig. 3b, pipelining graph iterations is allowed, which enables executing actors A and C from the second iteration in parallel with firings of actors D, F, and G from the first. The construction of efficient periodic schedules is an NP-complete optimization problem.

Memory allocation To support the execution of an HSDF graph on a target, each of its FIFOs must be allocated in memory or in a supporting hardware mechanism. Since, during a single iteration of an HSDF graph, each FIFO will be written to, and

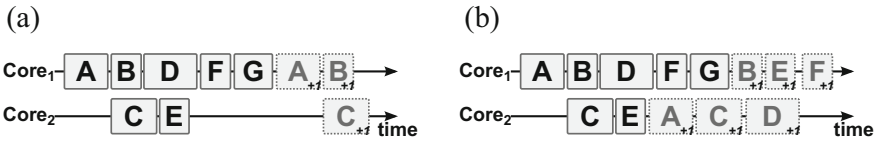


Fig. 3 Schedules of the HSDF from Fig. 2a on two cores. (**a**) Non-overlapping iterations. (**b**) Pipelining iterations

read from exactly once, the memory needed for allocating each FIFO corresponds to the size of one data token transiting through it. This assumption holds only if HSDF iterations are scheduled one at a time, which means that there are never two overlapping iterations scheduled concurrently. To save memory resources in shared memory MPSoCs, an address can be used to store several FIFOs, on the condition that two FIFOs that store tokens simultaneously may not be allocated in overlapping memory spaces (Desnos et al. 2015). For example, assuming that all tokens require exactly one memory slot and that tokens produced and consumed by an actor never exist simultaneously, the memory allocation presented in Fig. 4 can be derived for the HSDF graph from Fig. 2a.

The HSDF MoC is similar to the *task graphs* that can be found in many programming APIs, like OpenVX, DASK, or StarPU. In a task graph like in the HSDF MoC, each vertex represents a task to execute, and edges represent scheduling dependencies between tasks. Often, task graphs are more restrictive than the HSDF MoC. For example, task graph APIs often require the graph to be acyclic, model only scheduling dependencies with edges instead of capturing the data transfers between tasks, or instantiate a task graph for a single execution instead of several iterations.

The HSDF MoC was originally introduced as a sub-case of the SDF MoC, which is presented in the next section.

Synchronous Dataflow (SDF)

The Synchronous Dataflow (SDF) MoC was introduced by Lee and Messerschmidt (Lee and Messerschmitt 1987) to model and optimize DSP applications on parallel hardware. The semantics and an example SDF graph are presented in Fig. 5.

Fig. 4 Memory allocation for the HSDF graph from Fig. 2a

memory slot 0: $\overrightarrow{AB}$, $\overrightarrow{BE}$, $\overrightarrow{EG}$
memory slot 1: $\overrightarrow{AC}$, $\overrightarrow{CE}$
memory slot 2: $\overrightarrow{AD}$, $\overrightarrow{DF}$
memory slot 3: $\overrightarrow{GB}$
memory slot 4: $\overrightarrow{DG}$

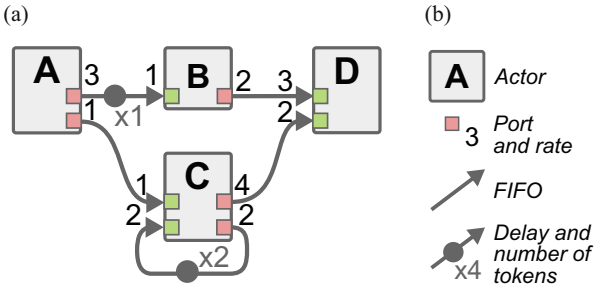


Fig. 5 Synchronous Dataflow (SDF) example and semantics. (**a**) SDF graph example. (**b**) SDF semantics

The only difference between the SDF and HSDF MoCs is that SDF actors may consume and produce more than one data token on each port at each firing. As illustrated in Fig. 5b, production and consumption rates are specified by an integer value written next to the ports. As in HSDF, SDF execution is data driven, meaning that an actor can fire as soon as it has at least as many tokens as specified by its consumption rates. Hence, it may happen that the number of tokens available on a FIFO is sufficient to trigger several firings of its consumer actor. An example of such behavior can be observed in Fig. 5a where actor A produces enough data tokens at each firing to trigger three firings of actor B.

Definition 1. An SDF graph $G = \langle A, F \rangle$ is a directed graph containing a set of actors A that are interconnected by a set of FIFOs F. $\text{prod}(f)$ and $\text{cons}(f)$ denote the actors in A producing and consuming tokens, respectively, on a FIFO f. For each output FIFO $f \in F$ connected to an actor $a \in A$, a data rate is specified by the function $\text{rate}_{\text{prod}} : A \times F \rightarrow \mathbb{N}^*$. Symmetrically, $\text{rate}_{\text{cons}} : A \times F \rightarrow \mathbb{N}^*$ defines the consumption rate of an actor $a \in A$ on an input FIFO $f \in F$.

Depending on the rates, the execution of some SDF graphs may always cause an indefinite accumulation, or the starvation, of data tokens in one or several FIFOs of the graph. An indispensable property for any valid system model is to avoid this inconsistent behavior. The *consistency* of an SDF graph can be defined mathematically by building its topology matrix.

Definition 2. Considering an SDF graph $G = \langle A, F \rangle$, the associated topology matrix Γ is a matrix of size $|F| \times |A|$ such that:

- Each column Γ_j of the matrix is associated with an actor $a_j \in A$ of G.
- Each row Γ_i of the matrix is associated with a FIFO $f_i \in F$ of G.
- The matrix coefficients are $\Gamma_{i,j} = \text{rate}_{\text{prod}}(a_j, f_i) - \text{rate}_{\text{cons}}(a_j, f_i)$.

The topology matrix associated with the SDF graph of Fig. 5a is presented hereafter. The columns and rows of the matrix are labeled with the corresponding actors and FIFOs, respectively.

$$\Gamma = \begin{array}{c} \\ \overrightarrow{AB} \\ \overrightarrow{AC} \\ \overrightarrow{CC} \\ \overrightarrow{BD} \\ \overrightarrow{CD} \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{pmatrix} 3 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & -3 \\ 0 & 0 & 4 & -2 \end{pmatrix} \end{array}$$

Note that $\Gamma(\overrightarrow{CC}, C) = 0$, since actor C produces and consumes the same number of tokens on its self-loop. In general, the production and consumption rates on self-loop FIFOs should always be equal. Otherwise, tokens will either accumulate indefinitely on this FIFO, or this FIFO will eventually cause a deadlock. In an SDF

graph, a deadlock occurs when the number of tokens in the FIFOs of the graph is not sufficient to enable any actor firing. Thus, an SDF graph $G = \langle A, F \rangle$ with topology matrix Γ is said to be *consistent* if and only if $\text{rank}(\Gamma) = |A| - 1$.

Theorem 1. *Consistency is a necessary condition for the existence of a periodic sequential schedule, without indefinite accumulation of tokens or starvation deadlocks.*

A proof for Theorem 1 can be found in Lee and Messerschmitt (1987). The consistency of an SDF graph implies the existence of a *repetition vector* of size $|A|$. The integer coefficients of a repetition vector give the minimal number of firings of each actor to return the graph back to its original state. Executing an iteration of an SDF graph consists of firing each actor of this graph as many times as given by the repetition vector.

Computing the repetition vector q of a topology matrix Γ consists of finding a positive integer vector that solves the following equation:

$$\Gamma \cdot q = \begin{pmatrix} 0 \cdots 0 \end{pmatrix}^T$$

The repetition vector for the SDF graph of Fig. 5a is $q = \begin{pmatrix} 1 & 3 & 1 & 2 \end{pmatrix}^T$.

The expressivity of the SDF MoC is equivalent to that of the HSDF model, meaning that any system modeled with one of the two MoCs can also be modeled with the other. This property is often exploited in order to derive an implementation from an SDF graph first by transforming the SDF graph into an equivalent HSDF graph (Pino et al. 1996) and then by using this HSDF graph to allocate computing and memory resources, as seen in section "Homogeneous Synchronous Dataflow (HSDF)". For example, Fig. 6 presents an HSDF graph that is equivalent to the SDF graph from Fig. 5a. This HSDF graph was obtained by duplicating each HSDF actor by their number of repetitions from the computed repetition vector. The so-called *fork* and *join* actors were introduced to distribute data from a unique producer actor to several consumers and respectively to gather data from several producer actors to a unique consumer. To complete the transformation, new data token types must be used, which combine multiple tokens from the original SDF graph to a single token in the HSDF graph.

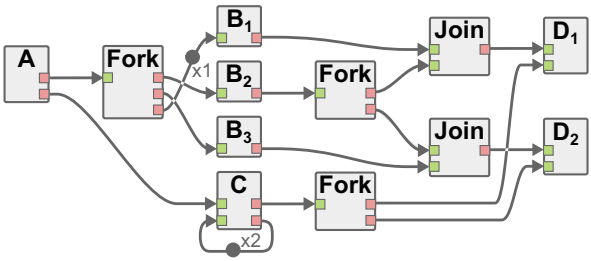


Fig. 6 HSDF derived from the SDF graph from Fig. 5a

The popularity of the SDF MoC comes from its ability to capture task and data parallelism, but also pipeline parallelism which can be explicitly specified using delays (Lee and Messerschmitt 1987). The multi-rate capabilities of SDF actors are also commonly used in signal processing applications for modeling functions with different input and output sampling rates, such as downsamplers or upsamplers. Finally, the static analyzability of the SDF MoC, and hence the possibility to map and schedule it at compile time, is a great asset for developing any systems where predictability is needed, such as real-time systems. The SDF MoC has been the topic of many research works, resulting in many analysis and optimization techniques to optimize their implementation both in hardware and software.

Further Static Extensions

Many extensions of the semantics of the SDF MoC have been proposed, each serving a different purpose. The following lists some of them:

- **Extended rate semantics:** To enable expressing more complex system behavior, the SDF semantics can be extended to allow more complex consumption and production patterns. For example, in the Cyclo-Static Dataflow (CSDF) MoC (Bilsen et al. 1996), each data port of an actor is associated with a finite list of integer production and consumption rates. CSDF actors iterate through the list of rates at each firing to determine the number of tokens produced and consumed on each port. The Affine Dataflow (ADF) MoC adopts a similar semantics where the list of rates associated with each port consists of a prologue, used once, and an iterative part (Bouakaz et al. 2012).
- **Hierarchy:** Some models introduce a hierarchical mechanism, enabling the specification of the internal behavior of actors with a dataflow subgraph. Such hierarchy mechanisms favor modularity and reusability of system models, making it easier to reuse parts of a system to build another, but also its dependability by making it easier to understand and maintain. A first hierarchy consisting of groups of SDF actors without additional rules of semantics was introduced in Lee and Messerschmitt (1987). In the absence of additional elements of semantics, groups of SDF actors may be ill-formed (Pino et al. 1996), resulting in hard to detect deadlocks. For this reason, more complex mechanisms have been introduced to enforce correct-by-construction graphs, e.g., in the Interface-Based SDF (IBSDF) (Piat et al. 2009) MoC.
- **Real-time extensions:** Extensions have been proposed to associate execution times with actor firings in order to better capture, analyze, and optimize real-time aspects of the modeled systems (Bouakaz et al. 2012).
- **Multidimension data:** Rates have also been extended to model multidimensional semantics (Keinert et al. 2005). For example, a 3D production rate of (n, m, p) means that a 3D array of n-by-m-by-p tokens is produced by the actor. Besides the production and consumption semantics, these extensions must also specify how multidimensional FIFOs behave. For example, if a 2D array of 10-by-10 tokens is available in a FIFO connected to an actor whose consumption rate is $(2, 2)$, there exist many different possible orders to consume the 2-by-

2 blocks: like raster scan, sawtooth scan, Z-order, or even with stencil-like consumption patterns (Keinert et al. 2005).

The HSDF and SDF MoCs, and all the aforementioned extensions, are static models. In static models, the consumption and production rates of actors are known at compile time and are independent of the value of data tokens at runtime. While this restriction enables powerful analysis and optimizations at compile time, it limits the expressivity of the models, preventing them from modeling data-dependent behavior. The dataflow MoCs presented in the next section alleviates this restriction.

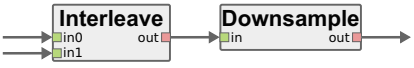
Dynamic Dataflow MoCs

The primary difference between static and dynamic dataflow MoCs is that the number of data tokens consumed and produced by actors is not known a priori in a dynamic model. Hence, instead fixed or periodic rates, a dynamic actor may dynamically change its exchange rates. Two of the most popular dynamic dataflow MoCs, KPN and DPN, are presented in the following.

Kahn Process Network

The Kahn Process Network (KPN) MoC was originally introduced in 1974 by Gilles Kahn. In the KPN MoC, there is no notion of atomic actor firings as in the HSDF and SDF models. Instead, each actor of the graph is associated with a continuously running *process* that can perform reads (pop) and writes (push) in the FIFOs connected to its ports. An example KPN graph with pseudocode of processes associated with each actor is given in Fig. 7. In this example, the *Interleave* actor uses an internal state variable `bool` to alternate readings from its two input ports and forwards the read token to its unique output. The *Downsample* actor reads two tokens with a non-null value before producing each output value.

The KPN MoC has a greater expressivity than the SDF model. The production and consumption of data may depend on an internal state of the actor, as is the case for the *Interleave* actor in Fig. 7, or may depend on the value of previously consumed tokens, as is the case for the *Downsample* actor. While this extra expressivity makes




```
process interleave (){
  static flag = false ;
  if ( flag ) x = in0 . read ();
  else x = in1 . read ();
  out . write (x); flag = ~flag ;
}
```

```
process downsample (){
  a = 0; b = 0;
  while (a == 0) a = in . read ();
  while (b == 0) b = in . read ();
  out . write ((a+b)/2.0);
}
```

Fig. 7 Kahn Process Network (KPN) example with pseudocode

KPN Turing complete (Buck 1993), and hence capable of modeling more complex systems, it also makes the model non-decidable. A dataflow model is said to be *decidable* if it is possible to derive a schedule for its computations at compile time. In the case of KPN, deriving a schedule is infeasible, since the computation and data exchange may depend on the value of data, which is known only at execution time.

An important characteristic of the KPN MoC is that, like the SDF MoC, it is a *deterministic* model. A MoC is said to be deterministic if the behavior of the controlled system, the history of tokens on every FIFO, and the outputs it produces solely depend on the inputs given to the system, independently from external factors such as time or randomness. In the KPN MoC, determinism is enforced by blocking read operations on FIFOs (Kahn and MacQueen 1976). Once initiated by a process, a read operation for N tokens will block the process in a waiting state until the requested tokens become available in the FIFO. The input FIFOs of a process can only be accessed using blocking reads, and it is not possible to *peek* at the number of tokens in a FIFO before reading from it.

Dataflow Process Networks

The Dataflow Process Network (DPN) MoC, introduced by Lee and Parks in 1995, has an almost identical semantics to KPN. Two notable differences are that DPNs do not have blocking reads and that actor computations are associated with a set of *firing rules*. A firing rule of a DPN actor is a set of conditions on the content of its input FIFOs, possibly referring to the number and values of tokens they contain. Once a firing rule is satisfied, the associated computations are executed atomically, possibly consuming and producing data tokens. Atomicity means that actor firing and associated production and consumption of tokens happen instantly from the MoC point of view.

Contrary to KPN, the DPN MoC is non-deterministic, and identical input data tokens may result in diverse outputs or behaviors. For example, let us assume that in the *Interleave* process of Fig. 7 the read operations may time out after a millisecond, automatically skipping to the read on the other input port of the actor. This valid behavior for a DPN actor may result in different outputs for the same inputs depending on when tokens become available in input FIFOs. If at $t = 0$, the following tokens are available in the input FIFOs, $\{5, 3, 7\}$ on port `in0`, and $\{2, 6, 1\}$ on port `in1`, the *Interleave* process will produce the following sequence: $\{5, 2, 3, 6, 7, 1\}$. Alternatively, if at $t = 0$, $\{5, 3, 7\}$ is available on `in0`, and only $\{2\}$ on `in1`, and the last two tokens, $\{6, 1\}$, become available on `in1` at $t = 10$ ms, the output will be $\{5, 2, 3, 7, 6, 1\}$. With the KPN semantics, identical output sequences would be obtained for both scenarios.

In the 2010s, the DPN MoC gained popularity in both the software and hardware programming communities. The CAL Actor Language (CAL) (Eker and Janneck 2003) for specifying DPNs can be translated into parallel software or hardware using frameworks such as ORCC (Yviquel et al. 2013) or OpenDF (Bhattacharyya et al. 2009).

Relation to Other Dataflow MoCs and Extensions

Being Turing-complete, dynamic models offer a greater expressivity than static models, which makes them simpler to understand for system developers. The price to pay for this flexibility is the lower predictability of dynamic dataflow MoCs, which often makes it impossible to verify system properties analytically, such as deadlock freedom, or memory boundedness. Moreover, because of data-dependent behaviors, many implementation aspects, such as the scheduling of computations, must be decided dynamically at runtime, which inevitably incurs an overhead on system performance.

As with static models, modifications of the DPN and KPN concepts can be found in the scientific literature. For example, the Scenario-Aware Dataflow (SADF) is akin to the DPN MoC, where firing rules are explicitly selected using a dedicated control flow based on a Markov chain (Stuijk et al. 2011). The main strength of the SADF MoC is the availability of analysis techniques for characterizing the latency, throughput, or memory consumption of systems.

Reconfigurable Dataflow

Reconfigurable dataflow is a third class of dataflow MoCs with a greater expressiveness than static models, but lower than dynamic ones. In reconfigurable dataflow MoCs, actor rates can be reconfigured at restricted points in the system execution. These restrictions make it possible to verify application properties, like schedulability, either at compile time or at runtime, after a reconfiguration occurred.

A reconfiguration semantics behind reconfigurable dataflow MoCs was originally developed by Neuendorffer and Lee (2004). This semantics is a mathematical model that makes it possible to detect potentially unsafe reconfigurations of an application graph. A reconfiguration is said to be unsafe if it may result in an unwanted and undetected state of the application, such as a deadlock or an inconsistency in the rates of actors.

The reconfiguration semantics (Neuendorffer and Lee 2004) is based on a hierarchy mechanism where the behavior of an actor may itself be defined with a dataflow graph. A firing of a hierarchical actor corresponds to an execution of its associated subgraph, which consumes and produces the number of tokens specified by the hierarchical actor rates. To ensure safety of the reconfigurations, the production and consumption rates of actors within a subgraph are allowed to change only when the encompassing hierarchical actor is quiescent, that is, between two firings. The model presented in the next subsection is an example of a dataflow MoC enforcing reconfiguration semantics.

πSDF

The semantics and a graph example of the Parameterized and Interfaced SDF (πSDF) MoC are presented in Fig. 8. The πSDF semantics combine the semantics of SDF, the hierarchy mechanism of the IBSDF MoC, and an explicit parameterization tree with a reconfiguration mechanism (Desnos et al. 2013).

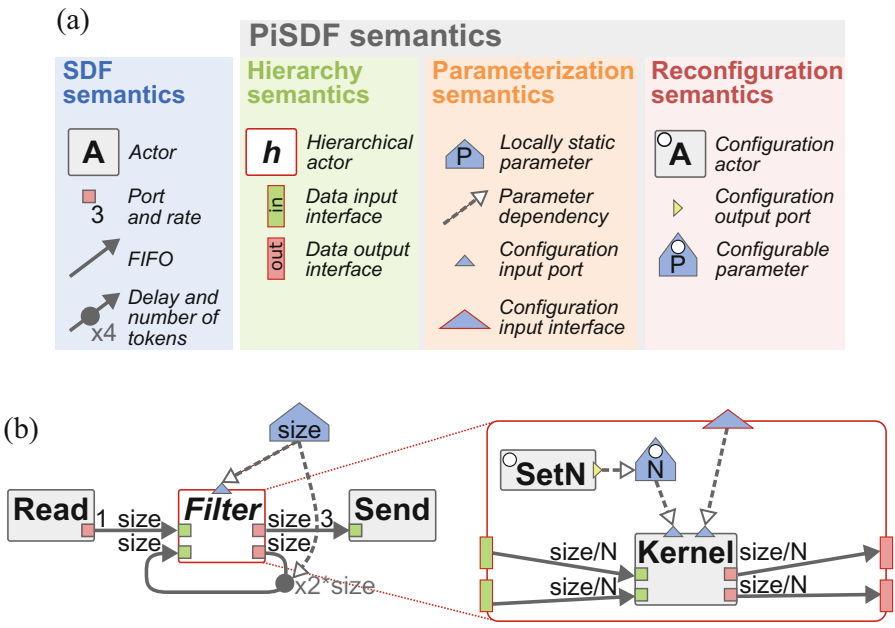


Fig. 8 Parameterized and Interfaced SDF (πSDF). (**a**) πSDF semantics. (**b**) πSDF graph example

In the πSDF MoC, each data port of a hierarchical actor is seen as a *data interface* to its subgraph. The purpose of an interface-based hierarchy (Piat et al. 2009) is to isolate the nested levels of hierarchy in terms of graph consistency analysis. In other words, the consistency of a hierarchy of graphs can be verified by analyzing separately the consistency of each of the underlying SDF graphs. To enable this compositionality, data interfaces automatically duplicate and discard data tokens if, during a subgraph iteration, the number of tokens exchanged on FIFOs connected to interfaces is greater than the number of tokens produced on the corresponding data ports of the parent actor.

The parameterization semantics of the πSDF MoC consists of a set of parameters P and parameter dependencies, configuration input ports, and interfaces. A parameter $p \in P$ is a vertex of the πSDF graph associated with an integer valuation function $\text{val} : P \rightarrow \mathbb{N}$. The value associated with a parameter is propagated through explicit dependencies to other parameters and to actors which may use this value in expressions specifying their own values, or rates of their dataflow ports. In the πSDF MoC, it is possible to disable all firings of an actor by setting all its rates to zero. As illustrated in Fig. 8b, parameter values can be propagated through multiple levels of hierarchy using a configuration input port on a hierarchical actor and a corresponding configuration input interface in the associated subgraph.

The reconfiguration semantics of the πSDF MoC is based on special configuration actors. When fired, reconfiguration actors are the only actors allowed

to dynamically change the value of a parameter in their graph. Reconfiguration actors must be fired exactly once per firing of their parent actor, before any non-configuration actor of their subgraph. This restriction is essential to ensure the safe reconfiguration of the subgraph to which configuration actors belong. For example, in the πSDF graph from Fig. 8b, when the hierarchical *Filter* actor is executed in the top-level graph, the *SetN* reconfigurable actor is the first one to be executed in its subgraph. When executing the *SetN* actor, a new value is set for parameter *N* which, in turn, is used to compute the production and consumption rates of the *Kernel* actor. When the value of parameter *N* is set, all rates in the subgraph of the *Filter* hierarchical actor are resolved, the repetition vector of the subgraph can be computed, and its consistency can be verified.

The short amount of time between a reconfiguration and the execution of actors can be used to verify graph properties, like graph consistency, but also to make on-the-fly resource allocation decisions. The SPIDER runtime supporting the execution and reconfiguration of πSDFs graphs on heterogeneous platforms will be presented in section "PREESM and SPIDER".

Other Reconfigurable Dataflow MoCs

As a more recent class of dataflow MoCs, reconfigurable dataflow models are less numerous than static or dynamic ones.

The Parameterized SDF (PSDF) model (Bhattacharya and Bhattacharyya 2001) is the first dataflow MoC to enforce the reconfiguration semantics from Neuendorffer and Lee (2004). The semantics of the PSDF, which inspired that of the πSDF MoC, separates each level of hierarchy into three subgraphs, each representing a different quiescent point for reconfiguration. A common point between πSDF and PSDF is that both models result from the application of a meta-model to the SDF MoC, the Parameterized and Interfaced dataflow Meta-Model (PiMM) (Desnos et al. 2013) and the parameterized dataflow meta-model (Bhattacharya and Bhattacharyya 2001), respectively. The purpose of a dataflow meta-model is to bring new elements to the semantics of a base dataflow MoC in order to increase its modeling capabilities. PiMM and parameterized dataflow have a similar purpose: to bring hierarchical graph composition and safe reconfiguration features to any dataflow MoC that has a well-defined notion of graph iteration and repetition vector, such as HSDF and CSDF, or multidimensional SDF (MDSDF) (Keinert et al. 2005). A base dataflow MoC whose semantics are enriched with PiMM or with the parameterized dataflow meta-model is renamed with prefixes π- and P-, respectively.

Another series of reconfigurable dataflow MoCs was proposed by Fradet et al. with the Reconfigurable Dataflow (RDF) (Fradet et al. 2018) and Boolean Parameterized Dataflow (BPDF) (Bebelis et al. 2013) MoCs. These MoCs were all proposed to enable specification of reconfigurable dataflow graphs whose consistency and liveness can be verified statically. In the πSDF MoC, the consistency of a (sub)graph can be verified only at runtime, after the configuration actors of this (sub)graph have set the values of all parameters, enabling the computation of the repetition vector. To enable compile time verification of the graph properties, the Synchronous Parameterized Dataflow (SPDF) and BPDF semantics impose rules

on when and how parameters of a graph can be reconfigured, without resorting to an explicit hierarchy mechanism (Bebelis et al. 2013). Using these rules and symbolic analysis that extends SDF analysis, the consistency of the graph can be verified at compile time, for any value dynamically taken by the parameters. In the RDF model, the reconfiguration semantics goes further by allowing changes in the topology of the graph itself (Fradet et al. 2018), using a dedicated graph transformation semantics. A dedicated compilation framework for the SPDF MoC was proposed by Dardaillon et al. in 2016.

The aforementioned reconfigurable MoCs use SDF as the underlying MoC. Another approach to achieve reconfigurability is to restrict the dynamism of a dynamic dataflow MoC, which was notably used in the Parameterized Set of Modes - Core Functional Dataflow (PSM-CFDF) MoC (Lin et al. 2015). In PSM-CFDF, the internal behavior of each actor is modeled using an FSM whose states dictate the actor rates. By jointly analyzing the FSMs of several connected actors at compile time, production and consumption patterns, called *modes*, can be identified for which it is possible to derive a schedule of actor firings or for which size of the FIFOs can be bounded. By breaking down a dynamic application into a set of predictable *modes*, each with predictable computation or memory access characteristics, the PSM-CFDF MoC makes it possible to perform smarter resource allocation on-the-fly, which is in general not feasible for dynamic dataflow MoCs. The switching from one mode to another can be seen as a reconfiguration of the application.

Optimization of Dataflow Programs

The previous section discussed different dataflow MoCs and how their properties make them highly useful to reason about the execution of a program. Tools can thus leverage the information in the model to optimize a program so as to improve its performance, energy efficiency, thermal dissipation, reliability or other execution metrics, or combinations thereof (Marwedel et al. 2011). The functional model of an application can also be extended to take constraints into account, like buffer sizing or precedence enforced by the scheduler. Depending on the model, this allows tools to reason about bounds on metrics while better reflecting the execution on the target system. This section describes optimization flows for dataflow programs, mostly for performance and energy efficiency. In particular, it explains different approaches for *mapping*, which refers to an automated process that decides where to place the elements of the model (nodes and edges) onto the target multicore. A mapping is often thought to account for both a *spatial* and a *temporal* placement. A spatial placement, for instance, specifies which core executes which node or which memory holds which edge. A temporal placement or scheduling, in turn, specifies in which order elements that share a resource should be executed.

Most optimization flows rely on models of the target multicore. After introducing such models, the discussion turns to different model-based mapping approaches that range from static or compile time to fully dynamic mapping at runtime. The terms static and dynamic are aligned with the taxonomy in Lee and Ha (1989).

Modeling Heterogeneous Platforms

As mentioned in section "Introduction", the complexity of MPSoCs continues to increase. Other sections and chapters of this book offer an excellent overview of the types of dedicated accelerators, emerging technologies, reconfigurable architectures, and application-specific processors being put forward by computer architects. The topology of a system and the characteristics of its components have to be captured for a tool to reason about the impact of mapping decisions on execution metrics. This is akin to latency and resource occupation tables in classical single core compilers. The following provides a, by no means exhaustive, overview of modeling approaches.

System-Level Description

Figure 9 shows a representative set of schematics of heterogeneous multicore SoCs. The popular 8-core big.LITTLE architecture as found in boards such as the Odroid-XU3/Odroid-XU4 (https://www.hardkernel.com/shop/odroid-xu4-special-price/) is depicted in Fig. 9a. The big cores, for instance, a Cortex-A15, and the little cores, for instance, a Cortex-A7, feature the same instruction set. In this case, heterogeneity stems from the different performance of the cores as a result of a different micro-architecture and clock frequency. The more recent DynamIQ technology from ARM allows integrating heterogeneous micro-architectures in a more flexible interconnect. Figure 9b shows the schematic of the Texas Instruments (TI) Keystone II architecture (Biscondi et al. 2012), integrating ARM cores and TI Very Large Instruction Word (VLIW) cores for digital signal processing. Mesh-based or tiled architectures can (cf. Fig. 9c) also be found in the market, like the coherent Neoverse N1 mesh from ARM (Pellegrini et al. 2020) or the Coolidge MPPA 2D torus from Kalray (de Dinechin 2015). These architectures use Networks-on-Chip (NoCs) as interconnect. For mapping optimization, tools use a description of topologies such as those shown in Fig. 9. This is usually implemented with ad hoc markup languages that describe the interconnect, the types of cores, and the memory subsystem. The level of detail varies from a pure schematic description, like in Sesame (Pimentel

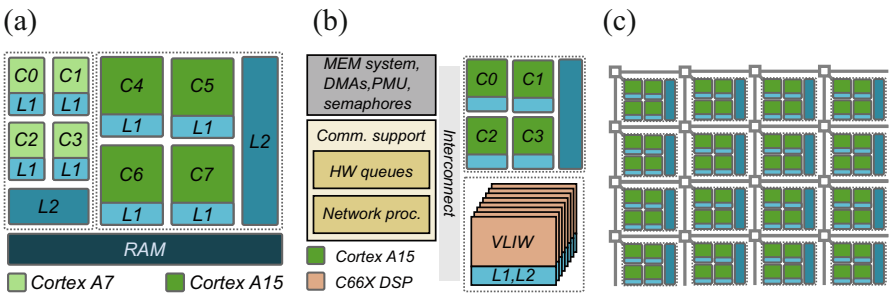


Fig. 9 Schematics of heterogeneous multicores. (**a**) ARM big.LITTLE. (**b**) TI Keystone II. (**c**) ARM CMN-600

et al. 2006), the Distributed-Object Layer (DOL) approach (Thiele et al. 2007), the S-LAM model (Pelcat et al. 2009), or Turnus (Brunet et al. 2013) to a deep modeling that includes the instructions and micro-architectural details of each of the cores (Eusse et al. 2013; C/DA Design Automation 2020).

Apart from the pure architectural description, tools for dataflow mapping have to understand the cost of the runtime environment that enforces the dataflow semantics. This includes possible communication primitives that can be used to implement the message-passing FIFO semantics of the high-level model. Some mapping tools can use these models to automatically select the best matching communication API in conjunction with the actor mapping (Castrillon et al. 2012). Similarly, information about multi-tasking or threading support, per-core scheduling policies, and associated costs are often modeled.

The effort on several such system-level modeling initiatives, including those of the MCA (The Multicore Association, Inc 2015) and the MPSoC Application Programming Studio (MAPS) project (Leupers et al. 2017), led to the recent IEEE 2804-2019 Standard for Software-Hardware Interface for Multi-Many-Core (C/DA Design Automation 2020). The standard defines an XML format to represent complex architectures, focusing on the information needed to evaluate the execution of software on them. This includes detailed information of cores (including VLIW), memory address spaces, caches, scratchpad memories, interconnection links, as well as communication and synchronization primitives. The more abstract system-level model used by Mocasin (Menard et al. 2021) is shown in Fig. 10 as example. With a simple Python syntax, shown on the left side of the figure, the script produces a model topology that corresponds to the ARM big.LITTLE system of Fig. 9a.

Inspired by formal MoCs, recent work focused on devising a mathematical model of the system-level architecture, called Model of Architecture (MoA) (Pelcat et al. 2016, 2018) (term and relation to MoCs first coined in Gerstlauer et al. 2009). This allows the mapping tool to directly use the system-level description of the target system in form of a MoA to evaluate the performance (or other metric) of an application described in a particular MoC. In this way, there is no need to resort to more detailed trace-based or full-system simulators as discussed in the following.

```
# Select processors from library
little_processor = Processor("PE", type="ARM_CORTEX_A7")
big_processor = Processor("PE",  type="ARM_CORTEX_A15")

# Add two cluster of processors
designer.addPeClusterForProcessor("cluster_a7", little_processor, 4)
designer.addPeClusterForProcessor("cluster_a15", big_processor, 4)

# Add L1 caches to each processor
designer.addCacheForPEs("cluster_a7", name='L1', <params>)
designer.addCacheForPEs("cluster_a15", name='L1', <params>)

# Add L2 caches to each cluster
designer.addCommunicationResource("L2_A7", ["cluster_a7"])
designer.addCommunicationResource("L2_A15", ["cluster_a15"])

# Add a RAM accessible by all PEs
designer.addCommunicationResource("DRAM",
   ["cluster_a7", "cluster_a15"])
```

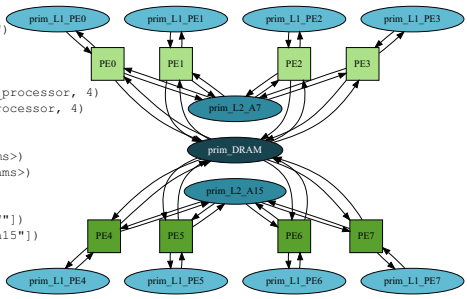


Fig. 10 Simplified model for a big.LITTLE architecture from Fig. 9a

Modular Performance Analysis (MPA) is another example of analytical models for performance estimation (Huang et al. 2012). MPA uses a refined analysis based on real-time calculus (Thiele et al. 2000), in which an edge of a dataflow graph is characterized by lower and upper bounds on the tokens that can be produced within a time window. By varying the window size, these bounds are treated as functions and are called *arrival curves*. By modeling resources and schedulers, MPA reasons about the execution of actors on a system as a process with service bounds, specifying the maximum and minimum token consumption rates from input channels.

Modeling Performance and Energy Consumption

Modeling the performance, or the energy/power consumption of code on a heterogeneous system, is an active area of research (cf. and ▸ Chaps. 25, "Processor Simulation and Characterization"; ▸ 27, "Virtual Prototyping of Processor-Based Platforms".) For dataflow applications, a mapping tool has to be made aware of the cost of executing nodes (actor firings or portions of code of a KPN node) and communicating data between nodes. The latter is usually captured by models of the interconnect or the communication primitives, where simple models have proven to be effective for shared-memory and mesh architectures (Tretter 2018; Lesparre et al. 2016). Modeling and estimating the processing time is considerably more complex (Ghasemi et al. 2021) (cf. ▸ Chap. 25, "Processor Simulation and Characterization"), since the internal operations of a dataflow node are not described explicitly in the MoC but is usually implemented in a mainstream imperative programming language. Cost models for processing and communication are needed to quickly evaluate a mapping solution, for which the focus of the mapping flows is on fast prototyping as opposed to cycle-accurate or instruction-accurate simulation. In addition to this, for mapping exploration it suffices to have a model with high fidelity irrespective of how accurate the model is. Fidelity guarantees that an estimated improvement due to a choice at exploration time would eventually lead to an improvement on the real system.

Compiler-based approaches leverage the information in the compiler intermediate representation along with profiling runs to estimate the cost of computation. This is related to the wealth of literature in Worst-Case Executing Time (WCET) analysis (Wilhelm et al. 2008). For average-case estimation, the code is instrumented and executed on the host. Virtual compiler backends can be used so that the instrumented version carries the cost of different target instruction set architectures (Gao et al. 2009; Eusse et al. 2013; Stemmer et al. 2020). A recent approach uses machine learning to model the throughput of basic blocks (Mendis et al. 2019). It remains to be seen how this approach can extend beyond basic blocks and how well it performs on specialized cores.

There are less works for estimation of energy or power consumption. Earlier Electronic System Level (ESL) models, based only on high-level events, showed relative good performance compared to instruction accurate models for embedded microprocessors (Van Stralen and Pimentel 2010) and systems on Field-Programmable Gate Array (FPGA) (Piscitelli and Pimentel 2011). Recent work has shown that linear models are good enough to model ARM-based systems (Schuermans and Leupers 2019; Pelcat et al. 2018). At a lower level, researchers have shown

that it is possible to reason about energy consumption at the level of the compiler intermediate representation (Georgiou et al. 2017). If the target system is available, performance and energy numbers can be measured and fed back to the mapping tool. This has been studied thoroughly in recent years for standard processors (Meneses-Viveros et al. 2021) and is well understood for embedded systems. The PAPIFY tool (Madronal et al. 2019), for instance, relies on performance counters of the architecture to measure the energy consumed by different parts of a system at runtime. By measuring clock cycles and cache misses for each actor firing, as well as an internal energy model, PAPIFY can automatically provide useful performance and energy metrics to improve the mapping of dataflow applications.

Based on cost models for computation and communication, different approaches can be used to model the performance of the entire application provided a mapping. Depending on the MoC, this can be done statically or with analytical models (cf. MoA mentioned above). For more dynamic models, trace-based simulators use the information in the system-level description and the costs associated with application nodes and edges to estimate the metrics for the parallel execution (Pelcat et al. 2014; Pimentel et al. 2006; Menard et al. 2021) (cf. ▸ Chap. 26, "Methodologies for Design Space Exploration"). Most of these trace-based simulators execute considerably faster than system-level simulators (cf. ▸ Chap. 27, "Virtual Prototyping of Processor-Based Platforms"). Properties of the MoC, such as determinism, legitimize trace-based simulation. This means, for instance, that small deviations in the real parallel schedule in the target system are guaranteed not to change the output of the application as observed during tracing (provided the same inputs, in the case of dynamic MoCs).

Static Mapping

Finding a static mapping for a MoC-based program onto a fixed heterogeneous multicore is a special case of the general formulation for Design Space Exploration (DSE) of ▸ Chap. 26, "Methodologies for Design Space Exploration". To find a mapping, the target system is modeled as a multi-graph which contains cores and communication links $S = (C, L)$ (Simpler and earlier formulations assume a fully connected architecture and thus do not require modeling communication links.). Cores can be annotated with types, supported accelerated routines, and supported runtime configurations, while a link can be annotated with its bandwidth, latency, and involved resources (Direct Memory Access (DMA), memories, buses, or NoC links). Given a dataflow program represented as a graph $G = \langle A, F \rangle$, a static mapping is a fixed allocation of the application elements to the resources of the platform. In other words, the allocation is decided ahead of time and does not change during the execution of the application. Often, the mapping process focuses on mapping the nodes of the application, that is, its goal is to find $\mu_A^* : A \to C$ that minimizes a set of system metrics. For instance, if optimizing for the maximum throughput t_a for a given node $a \in A$, as single metric, then $\mu^* = \text{argmax}_{\mu \in C^A} t_a(\mu)$. More complete formulations compute a mapping for the edges $\mu_F : F \to L$, fix free parameters of

the application like buffer sizes, include constraints for mapping to accelerators, and compute an order for elements that are mapped to the same resource. Note that such formulations impose implicit constraints on valid solutions. For instance, a mapping of a channel $f = (a_i, a_j) \in F$, $\mu_F(f) = l \in L$ is valid only if the cores resulting from the mapping $\mu_A(a_i) = c_k \in C$ and $\mu_A(a_j) = c_m \in C$ are indeed connected via the link l, that is, $l = (c_k, c_m) \in L$. Similarly, an edge mapping must respect the size constrains of the underlying resources of a selected link, for instance, the size of the physical memories.

Figure 11 shows a generic static mapping flow. As inputs, the mapper takes the application model, the architecture model, and constraints. Constraints can represent real-time requirements, such as a given throughput for an actor, or a latency constraint over a path in the graph. Constraints can also define prescribed mappings for actors, in the case an actor can only be mapped to an accelerator, or a predefined subset of resources that implement a given functionality, like a mapping to a set of cores with access to a particular peripheral. Often the mapper iterates multiple times before finding a suitable solution that is then passed as the final result (μ_A^*, μ_F^* in the figure).

For static dataflow models, a common approach to compute a mapping consists in analyzing one or several unrolled iterations of the graph (recall from sections "Homogeneous Synchronous Dataflow (HSDF)" and "Synchronous Dataflow (SDF)"). In the SDF case, this amounts to an HSDF transformation, unfolding the resulting HSDF graph and removing edges with delays. The resulting graph is a directed acyclic graph for which a wealth of methods exist to compute a mapping (Kwok and Ahmad 1999). The schedule resulting from a so-computed mapping is called a block schedule, since iterations of the unrolled graph do not overlap in time during the execution of the application. Algorithms also

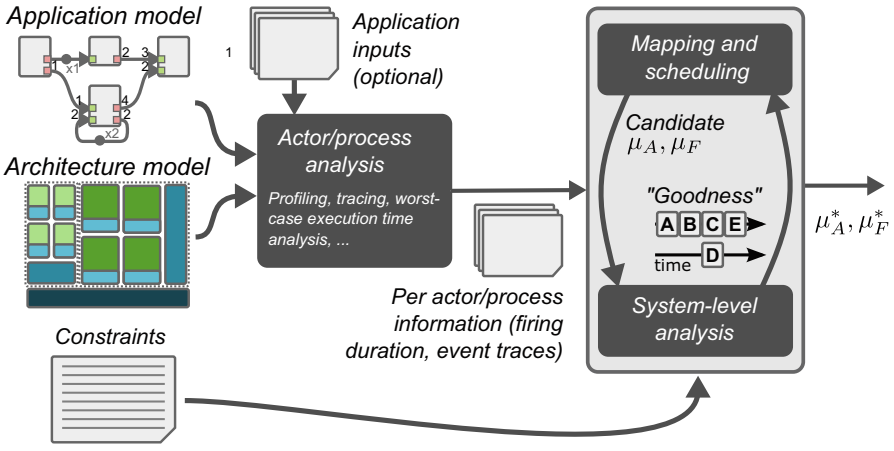


Fig. 11 Generic static mapping flow

exist for computing overlapped schedules and thus exploit graph-level pipeline parallelism (Honorat et al. 2020).

Although the nature and order of computations in dynamic dataflow MoCs are, by definition, data-dependent, the mapping of computational nodes on processing elements, be they actors or processes, is often done statically. The scheduling decisions, however, which order computations on the different processing elements, are mandatorily taken at runtime for such models. For KPNs and other dynamic dataflow models, representative runs of the application are required for the mapper to understand how processes in the application exchange information with one another. Each application run is modeled via traces, which record the read and write events on the channels (Castrillon et al. 2010; Van Stralen and Pimentel 2010). The traces can be used directly to replay the application behavior on a high-level discrete event simulator (Pimentel et al. 2006; Castrillon et al. 2013; Brunet et al. 2013) or can be represented as a graph (Castrillon et al. 2012; Brunet 2015). For the latter, edges model production-consumption relationships, buffer sizing constraints, and guarded actions in dynamic dataflow actors among others. A graph representation enables defining quick heuristics that does not require replaying the traces on a simulator (Castrillon et al. 2013; Brunet 2015).

Apart from heuristics crafted for the special purpose of solving the static mapping problem, several meta-heuristics have been adapted to solve the problem as well. A meta-heuristic is a generic solution approach that can be reused and adapted for particular problem formulations. Genetic algorithms, for instance, mimic the process of evolution in a pool of solutions to a given problem. Each solution, or *individual*, is encoded as a string or *chromosome*. The pool of solutions is evolved over generations by using operators for mutation (of a single individual) and crossover of two individuals, which mimics the reproduction process. ▸ Chapter 26, "Methodologies for Design Space Exploration" goes into much more detail about how this process can be further customized to solve the static mapping problem. Other meta-heuristics include random walk, simulated annealing (Kirkpatrick et al. 1983), and tabu search (Glover 1989).

In practical frameworks, the mapping problem cannot be fully decoupled from the implementation. Actor firings can be implemented by means of a task runtime that distributes ready tasks to worker threads in a way predefined by the computed mapping (cf. ▸ Chap. 30, "Parallel Programming Models" for modern task-based runtime systems). Actor firings can also be statically scheduled within threads mapped to cores or managed freely by the operating system (Hautala et al. 2018). Alternatively, an actor or a process in a KPN can be mapped to a persistent thread (one process per thread), with the mapping enforced via pinning. A combination of these implementations is also possible, which further complicates the analysis and optimization of mappings. Similar implementation options exist for communication channels, which can be implemented by true message passing, by implementing FIFO buffers in shared memory, or by a custom memory pool that allows for memory reuse between logical buffers. In addition, as mentioned in section "Dataflow Models of Computation", techniques exist to reuse memory space for multiple channels for dataflow (Desnos et al. 2015) and for KPNs (Tretter 2018).

Hybrid Mapping

Static mapping approaches can lead to an efficient program execution, especially for regular applications like those described with static dataflow. A static mapping can outperform a dynamic mapping at runtime, since the static mapper leverages the information in the model about interaction patterns among actors, like rates in an SDF. In addition to this, the static mapper incurs less scheduling and synchronization overhead at runtime. Better dynamic mappings may exist, but a system may encounter them only rarely, leading to a high variation in the execution of an application using dynamic mapping. The time predictability of static mappings is another reason why these mappings have been studied so thoroughly over the past decades.

Hybrid mapping approaches seek to combine the flexibility of a fully dynamic mapping while retaining the benefits of static mappings (efficiency and time predictability). The flexibility of a dynamic mapping became a must as embedded systems ceased to be designed for a single task. Many embedded systems today are designed to execute a mix of applications, like sensor fusion, image acquisition and enhancement with artificial intelligence, and a communication stack for wireless transmission. Hybrid mapping consists of a time-consuming static mapping phase and a lightweight dynamic mapping phase (cf. Fig. 12). The static phase may compute partial solutions that are completed at runtime or may compute candidate variants for the runtime system to choose from. A partial solution can be represented as a set of constraints that restricts the search space for the runtime mapping. Multiple candidate variants are often selected from a Pareto front that combines two or more system metrics. For instance, a variant could be fast but energy hungry, or slow and energy efficient, or any dominating trade-off between performance and energy consumption. At runtime, the dynamic mapping phase would complete a

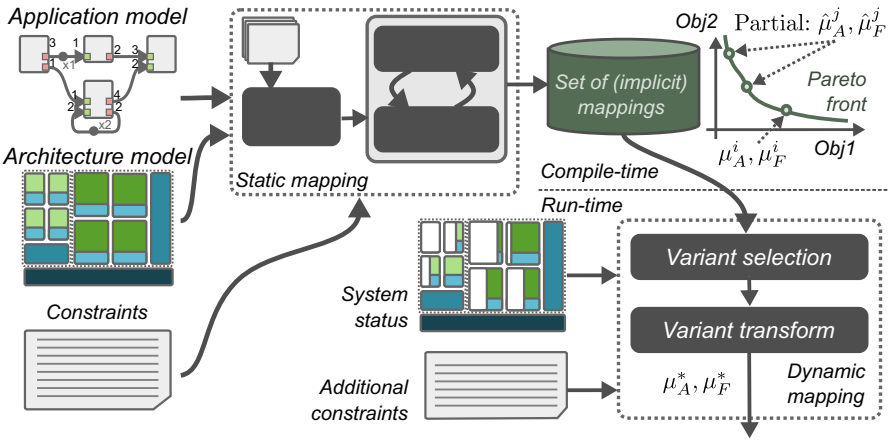


Fig. 12 Generic hybrid mapping flow

partial solution or select a mapping candidate by taking into account the current state of the system. The state of the system consists in a description of the resource occupation of the system at application launch time (this is represented as white boxes in Fig. 12). Apart from the system status, the dynamic phase may also receive additional constraints for a particular execution of the application. These constraints help in selecting a subset of the Pareto front described by the variants.

The work in Weichslgartner et al. (2018) is a good representative of a hybrid mapping methodology which was developed in the context of the Collaborative Research Center "Invasive Computing" (Teich et al. 2011). The authors use a hybrid mapping approach for applications represented as task graphs to increase the utilization of a multicore while respecting real-time constraints and saving resources. In this particular work, static mapping decisions are encoded as a set of constraints. A constraint can, for instance, specify what kind of cores should execute a task (by delivering an execution time below a bound), or at most how *far* away two tasks can be mapped. Distance, in this case, can refer to the number of hops for different routes in a NoC-based architecture. The constraint representation allows for resource sharing (processors and network links), as opposed to strict spatial isolation, which increases the utilization of the system as a whole. At runtime, a constraint solver can be used to determine the final mapping. The authors also propose a faster backtracking heuristic to reduce the complexity of the runtime mapping phase.

The works in Singh et al. (2011), Quan and Pimentel (2015), and Goens et al. (2017a) are good examples of methodologies that pre-compute a set of candidate mappings that are later selected at runtime. In its simplest form, the runtime mapper consists of a look-up for the mapping that meets the requirements and fits in the available resources. This requires less effort than solving a set of constraints but, naturally, will fail more often since the mappings remain quite rigid. To relax this, the work in Goens et al. (2017a) relies on the exploitation of platform and application *symmetries*, as formalized in Goens et al. (2017b). A symmetry refers to a transformation that can be applied to a static mapping without changing the characteristics of the mapping, that is, two symmetric mappings should lead to the same performance and energy consumption. Symmetries allow clustering static mappings into equivalence classes, reducing the search space at compile time, and enabling transformations at runtime. The symmetry-aware runtime system achieved performance on par with the Linux Completely Fair Scheduler (CFS) scheduler while reducing the time variation by two to three orders of magnitude. Symmetries are more restrictive than generic constraints, since they require mappings to be equivalent as opposed to be either equal or better. Efficient algorithms based on symmetries make it possible to switch mapping configurations during the execution of an application to reduce the overall energy consumption while respecting real-time constraints (Khasanov and Castrillon 2020; Khasanov et al. 2021).

Another kind of hybrid mapping strategy for dynamic dataflow MoCs consists of first using a statically defined mapping and using runtime-collected metrics to update this mapping sporadically (Yviquel et al. 2017).

Examples: Models and Tools

After having reviewed the fundamentals of dataflow and mapping optimization of dataflow programs, this section now reports on existing frameworks that use dataflow as an underlying MoC. It first provides an overview of tools in which the dataflow programming model is used, without end users necessarily being aware of it. The section then provides a greater level of detail on two academic tools, reporting recent extensions. This should convey an idea of current trends in the research of MoCs and optimizations for multi- and many-cores.

Dataflow in Commercial and Mainstream Tools

Dataflow models in different flavors underlie several existing tools. Examples are the Signal Processing WorkSystem (Synopsys Signal Processing WorkSystem (SPW) 2013), originally from Cadence and now with Synopsys, or Synopsys System Studio (2010) originally from CADIS. The designers of these tools explicitly restricted the expressivity of the model to ensure deterministic execution, thus allowing users to focus on the algorithmic aspects of signal processing. The NI LabVIEW Communications System Design Suite also relies on dataflow to model communication algorithms, restricting the more expressive language of general NI LabVIEW. Matlab Simulink also uses a model that, at the surface, resembles dataflow. The semantics of Simulink includes time-triggering and thus goes beyond pure dataflow, but a mapping to dataflow has been reported in the literature (Klikpo et al. 2016). These tools are mostly for modeling and specification, and, while there are tool flows to generate code, the common practice is to reimplement the algorithms once they are tested on a lower-level programming model. Kalray, a fabless many-core technology company, used ΣC (Goubier et al. 2011) as a programming language for their MPPA platform (de Dinechin 2013). ΣC allowed specifying CSDF programs. The principles of dataflow are also found in libraries, programming frameworks, and programming models, such as in OpenVX or OpenMP4x (cf. ▸ Chap. 30, "Parallel Programming Models"). Dataflow models are also used in high-level synthesis tools, for instance, by Maxeler (Koliogeorgi et al. 2019).

MPSoC Application Programming Studio (MAPS)

The MAPS project started around 2007 at the RWTH Aachen University, originally as a trace-driven auto-parallelizing compiler (Ceng et al. 2008). Over the years, the framework was extended with a high-level simulator for parallel execution (Ceng et al. 2009), support for parallel applications (Leupers and Castrillon 2010), mapping heuristics for multiple concurrent applications (Castrillon et al. 2010), support for hardware accelerators (Castrillon et al. 2011), and support for multiple backends and ESL simulators. Today, the ideas in MAPS continue to be developed

academically at the RWTH Aachen and the TU Dresden universities and commercially at Silexica GmbH/Inc (Leupers et al. 2017).

For parallel applications, MAPS defined the language C for Process Networks (CPN) to describe parallel applications using the KPN MoC. CPN is an extension to the C language with keywords for KPN processes, SDF actors, channels, and parameters (Castrillon and Leupers 2014) (akin to the example in Fig. 7). As discussed in section "Static Mapping", given the dynamic MoC, mapping heuristics in MAPS rely on traces following a flow similar to that in Fig. 11. With detailed cost models of the TI DSPs, trace-based mapping heuristics managed to obtain results comparable to those obtained via manual optimization by expert programmers as reported in Leupers et al. (2017). A comparative analysis of the quality of the mappings obtained with simple heuristics and genetic algorithms for KPN applications can be found in Goens et al. (2016).

Figure 13 shows an updated version of this comparison on a heterogeneous ARM big.LITTLE platform as described in Fig. 9a. The heuristics used are the Group-Based Mapping (GBM) heuristic (Castrillon et al. 2012) and a static variant of the Linux CFS. The metaheuristics include a random walk, simulated annealing (Orsila et al. 2007), tabu search (Manolache et al. 2008), and genetic algorithms (Erbas et al. 2006). These mapping algorithms were executed ten times on each application of the Embedded System Synthesis Benchmark Suite (E3S) benchmarks (Dick 2008), which were adapted to multicore systems using the method presented by Schwarzer et al. (2017). The figure reports the (geometric) means of the relative times, both for the execution time of the application with the generated mapping and the exploration time required. Unsurprisingly, metaheuristics can produce better mappings but require considerably more computational time.

The situation changes when the complexity of the platform increases. Figure 14 shows the same experiments, this time on a model of the Kalray MPPA3 Coolidge (Kalray Inc 2020). This platform consists of 5 identical clusters fully connected in a NoC, where each of the clusters has 16 identical cores, as well as specialized secure and management cores. The difference in performance of

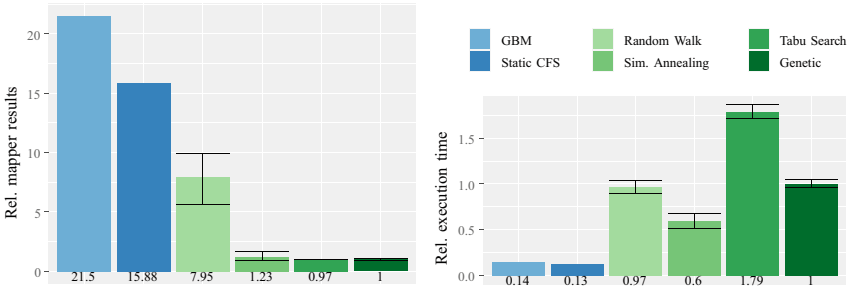


Fig. 13 Comparison of multiple mapping heuristics and metaheuristics on the Odroid-XU4 platform based on ARM big.LITTLE for the E3S benchmark suite

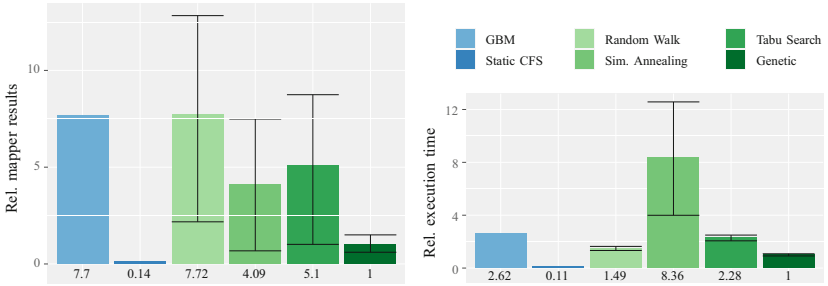


Fig. 14 Comparison of multiple mapping heuristics and metaheuristics on the Kalray MPPA3 Coolidge platform for the E3S benchmark suite

heuristics and metaheuristics in these larger platforms is less marked. Notably, the static CFS heuristic significantly outperforms every other algorithm, both in terms of mapping quality and exploration time. More sophisticated metaheuristics have more trouble with extremely large design spaces. Indeed, for the largest application in the E3S benchmarks, the design space has $85^9 \approx 2.3 \cdot 10^{17}$ mappings, which is more than 10^9 times larger than the design space for the big.LITTLE architecture above.

More recently, approaches based on exploiting the symmetry of the problem (cf. section "Hybrid Mapping") have helped in reducing the size of design spaces (Goens et al. 2017b; Schwarzer et al. 2017). Figure 15 shows the same setup as above, evaluating the E3S benchmarks on the same two platforms. In the figure, a standard variant of the algorithms is compared to one where the design space is pruned using symmetries. In terms of exploration time, using symmetries tends to reduce the time by a considerable amount. More importantly, however, they seem to mitigate the poor performance of the metaheuristics on the more complex design space of the MPPA3 Coolidge. The same simulated annealing heuristic performs an *average* of 32× better on this symmetry-pruned design space.

In a recent effort, these newer developments as well as a subset of the methods described in section "Optimization of Dataflow Programs" were combined into the Mocasin toolbox (Menard et al. 2021) and released under an open-source license (Mocasin GitHub Repository: https://github.com/tud-ccc/mocasin.). Mocasin provides a high-level simulator for performance estimation and implements various mapping strategies, including the symmetry-based and hybrid approaches discussed above. Mocasin does not provide a complete tool flow from source code to an optimized implementation tailored for a specific platform. Mocasin, instead, is a tool designed specifically to support researchers in their effort of developing better strategies for mapping complex applications to heterogeneous many-cores. Therefore, Mocasin is designed as a flexible infrastructure with increased interoperability with other tools, allowing for quick prototyping and evaluation of new approaches. Since Mocasin already implements a wide range of known mapping strategies, new methods can be directly compared to the state of the art in a comparison similar to the one shown in Figs. 13 and 14. The authors expect Mocasin and similar

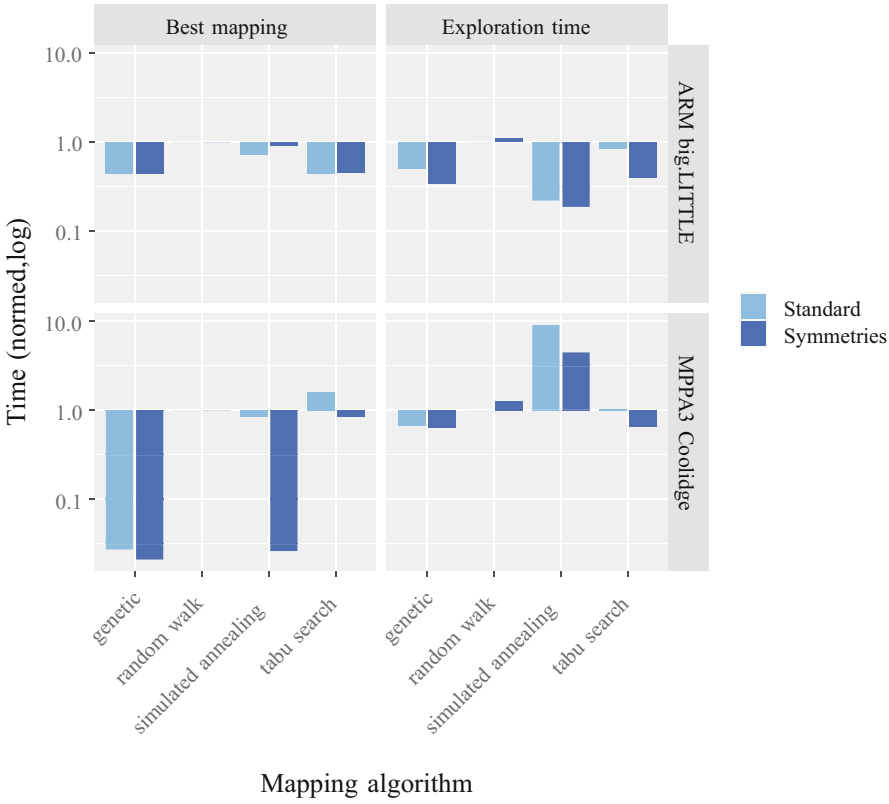


Fig. 15 Improving design space exploration via symmetries

tools to help consolidate years of results, enable reproducibility of results, and thus help accelerate research in this domain. The framework is currently being extended to support the richer reactor MoC (Lohstroh et al. 2020), which has proven instrumental to reason about time-deterministic execution of reactive automotive applications (Menard et al. 2020).

PREESM and SPIDER

Parallel and Real-time Embedded Executives Scheduling Method (PREESM) (Pelcat et al. 2014) and Synchronous Parameterized and Interfaced Dataflow Embedded Runtime (SPIDER) (Heulot et al. 2014) are two dataflow programming tools developed by the Institut d'Electronique et des Technologies du numéRique (IETR). The two tools rely on the πSDF MoC, presented in section "πSDF", for modeling applications. The main difference between the two tools is that PREESM focuses on purely static applications, while SPIDER is a runtime manager for dynamically reconfigurable applications.

Preesm

Preesm, which stands for Parallel and Real-time Embedded Executives Scheduling Method, is an open-source rapid prototyping framework created in 2007 at the IETR, in collaboration with Texas Instruments (Pelcat et al. 2014). Preesm is developed as a set of plugins for the Eclipse integrated development environment. As a rapid prototyping framework, the purpose of Preesm is to enable developers to design an application and to rapidly assess Key Performance Indicators (KPIs) of its deployment on a given heterogeneous MPSoC. For example, KPIs optimized and predicted by Preesm are computation and communication resource utilization (Pelcat et al. 2009, 2016), application throughput and latency (Pelcat et al. 2009; Deroui et al. 2017), energy consumption (Pelcat et al. 2016; Holmbacka et al. 2014), or memory footprint (Desnos et al. 2015, 2016). In addition to predicting these metrics, Preesm also offers software synthesis capabilities for generating working prototypes on commercial multi- and many-core chips (Pelcat et al. 2014; Hascoët et al. 2017).

The workflow adopted by Preesm is the Y-chart methodology (Kienhuis et al. 2001) depicted in Fig. 11. The three inputs taken by Preesm are:

- A dataflow description of the application using the πSDF model, stored in a derivative of the GRAPHML format. Preesm integrates a graphical editor for intuitively composing hierarchical πSDF graphs. As an alternative, the Higher Order dataflow Coordination Language (HoCL) can be used to describe πSDF graphs (Sérot 2020). HoCL is a language dedicated to the description of dataflow graphs but is not tied to a particular semantics. HoCL can be used to describe hierarchical and parameterized graphs using either structural or functional programming. Using an ingenious annotation system, HoCL supports many dataflow MoCs: HSDF, SDF, PSDF, πSDF, and DPN to cite just a few.
- A high-level description of the targeted architecture using the System-Level Architecture Model (S-LAM) (Pelcat et al. 2009). The purpose of the S-LAM description of the architecture is to give only the needed information for the design space algorithms run by Preesm to be successful.
- A scenario defining all the information needed for deploying a given application on a given architecture. Indeed, the πSDF graph and the S-LAM model being agnostic of each other, a third input is needed in Preesm to define all deployment constraints specific to a given pair of application and architecture. As such, the scenario defines mapping constraints, individual actor execution times for the different types of cores in the architecture, and energy consumption per actor per type of core.

From these inputs, Preesm successively performs graph transformations to flatten and unroll the input πSDF graphs into HSDF equivalents. Then, the graphs are mapped and scheduled onto the heterogeneous cores of the target architecture, and memory is allocated in the shared (Desnos et al. 2015) and distributed (Desnos

et al. 2015) memory banks. Finally, multi-threaded C code is generated, translating the resource allocation decisions made by the tools into executable code.

PREESM capabilities have been tested on a wide variety of targets, including multi-core general purpose processors, embedded heterogeneous ARM Big.LITTLE architecture, Texas Instruments Keystone I and II multi-core digital signal processors, and Kalray's many-core MPPA architecture. A collection of state-of-the-art open-source streaming, computer vision, and signal processing applications is available on the GitHub repository (PREESM GitHub Organization: https://github.com/preesm/) of PREESM.

SPIDER

In PREESM, all mapping, scheduling, and memory allocation decisions are made statically at compile time, which forbid the use of reconfigurable capacities of the πSDF MoC. In order to run a dynamically reconfigurable πSDF graph, a runtime manager is needed to handle graph reconfigurations and to manage resource allocation on-the-fly while executing the application. The Synchronous Parameterized and Interfaced Dataflow Embedded Runtime (SPIDER) (Heulot et al. 2014) serves that purpose for reconfigurable πSDF graphs.

The inputs and workflow of SPIDER is identical to that of PREESM, illustrated in Fig. 11, with two major differences. (1) Values of reconfigurable parameters of the πSDF graph are set dynamically by configuration actors of the application, and as a result, (2) all graph transformation and resource allocation decisions are made at runtime. Using a runtime manager to control the execution of a reconfigurable graph has an overhead on application performance. Indeed, such runtime manager requires processor time to compute repetition vectors, to perform graph transformations, and to map and schedule actor firings. Nevertheless, as presented in Heulot et al. (2013, 2014), even with large reconfigurable graphs with several hundreds of actors, this overhead is largely compensated by the efficiency of the scheduling decisions. The performance of SPIDER proved to be on par with, or better than, OpenMP performance for suitable applications (Heulot et al. 2014). As for PREESM, the performance of SPIDER was assessed on a wide variety of signal and image processing applications and on several commercial heterogeneous multi- and many-core targets.

In recent development, leading to the release of SPIDER 2.0 (SPIDER 2.0 GitHub Repository: https://github.com/preesm/spider2), the efficiency of the SPIDER for mapping and scheduling πSDF graphs was greatly improved by using a numerical representation instead of performing the πSDF to HSDF transformation (Arrestier et al. 2019). By using a numerical representation instead of building and storing HSDF graphs for resource allocations, the memory footprint of the runtime manager was reduced on average by 97%, and the overhead of the runtime manager was reduced on average by 85%.

Conclusion and Outlook

This chapter discussed how dataflow MoCs and MoCs in general are appealing alternatives to get a handle on the complexity of programming modern MPSoCs. It provided definitions and examples of the most prominent dataflow MoCs, including HSDF, SDF, and KPN, and gave insight into more recent adaptable models such as πSDF. The chapter also explains how the properties of a MoC can be leveraged to engineer tool flows that, based on models of the target system, produce an optimized implementation of the program. Static and hybrid optimization approaches were discussed and illustrated by means of two academic tools with more than a decade of research, MAPS and PREESM. Moving forward, the authors expect efforts to consolidate into open-source tools like PREESM, Ptolemy (Ptolemaeus 2014) and Mocasin (Menard et al. 2021). Interoperability and fast prototyping of models and algorithms will speed up advances in the field. Such advances are dearly needed, with ever more dynamic workloads coming up in interconnected emerging applications like autonomous driving and 5G communication. Similarly, emerging technologies (Castrillon et al. 2018) and interconnect (Fettweis et al. 2019) will require a more principled approach to program synthesis, enabled in part by future MoCs.

Acknowledgments This work was funded in part by the German Federal Ministry of Education and Research (BMBF) through the E4C project (16ME0426K), by the BMBF project 6G-life hub (16KISK001K), by the German Research Foundation (DFG) through TraceSymm (366764507), by the Studienstiftung des Deutschen Volkes, by the CERBERO (Cross-layer modEl-based fRamework for multi-oBjective dEsign of Reconfigurable systems in unceRtain hybRid envirOnments) Horizon 2020 Project, by the European Union Commission under Grant 732105, and by the French Agence Nationale de la Recherche under grant ANR-20-CE46-0001 (DARK-ERA project).

References

Alur R, Courcoubetis C, Dill D (1990) Model-checking for real-time systems. In: [1990] Proceedings. Fifth annual IEEE symposium on logic in computer science. IEEE, pp 414–425

Arrestier F, Desnos K, Juarez E, Menard D (2019) Numerical representation of directed acyclic graphs for efficient dataflow embedded resource allocation. ACM Trans Embed Comput Syst 18(5s). ISSN: 1539-9087. https://doi.org/10.1145/3358225

Bebelis V, Fradet P, Girault A, Lavigueur B (2013) BPDF: a statically analyzable dataflow model with integer and boolean parameters. In: 2013 proceedings of the international conference on embedded software (EMSOFT). IEEE, pp 1–10

Bhattacharya B, Bhattacharyya SS (2001) Parameterized dataflow modeling for DSP systems. IEEE Trans Sig Process 49(10):2408–2421

Bhattacharyya SS, Brebner G, Janneck JW, Eker J, Von Platen C, Mattavelli M, Raulet M (2009) OpenDF: a dataflow toolset for reconfigurable hardware and multicore systems. ACM SIGARCH Comput Archit News 36(5):29–35

Bilsen G, Engels M, Lauwereins R, Peperstraete J (1996) Cycle-static dataflow. IEEE Trans Sig Process 44(2):397–408

Biscondi E, Flanagan T, Fruth F, Lin Z, Moerman F (2012) Maxi-mizing multicore efficiency with navigator runtime, White Paper. www.ti.com/lit/wp/spry190/spry190.pdf

Bouakaz A, Talpin J-P, Vitek J (2012) Affine data-flow graphs for the synthesis of hard real-time applications. In: 2012 12th international conference on application of concurrency to system design. IEEE, pp 183–192

Brunet SC (2015) Analysis and optimization of dynamic dataflow programs. PhD thesis, Ecole Polytechnique Federale de Lausanne (EPFL)

Brunet SC, Alberti C, Mattavelli M, Janneck J (2013) Turnus: a unified dataflow design space exploration framework for heterogeneous parallel systems. In: 2013 conference on design and architectures for signal and image processing (DASIP), pp 47–54

Buck JT (1993) Scheduling dynamic dataflow graphs with bounded memory using the token flow model. PhD thesis, EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/1993/2429.html

Castrillon J, Leupers R (2014) Programming heterogeneous MPSoCs: tool flows to close the software productivity gap. Springer, p 258. ISBN: 978-3-319-00675-8. https://doi.org/10.1007/978-3-319-00675-8

Castrillon J, Velásquez R, Stulova A, Sheng W, Ceng J et al (2010) Trace-based KPN composability analysis for mapping simultaneous applications to MPSoC platforms. In: Proceedings of the conference on design, automation and test in Europe DATE'10. European Design and Automation Association, Dresden, pp 753–758. ISBN: 978-3-9810801-6-2. http://dl.acm.org/citation.cfm?id=1870926.1871107

Castrillon J, Schürmans S, Stulova A, Sheng W, Kempf T et al (2011) Component-based waveform development: the nucleus tool flow for efficient and portable software defined radio. Analog Integr Circuits Sig Process 69(2–3):173–190. ISSN: 0925-1030. https://doi.org/10.1007/s10470-011-9670-1

Castrillon J, Tretter A, Leupers R, Ascheid G (2012) Communication-aware mapping of KPN applications onto heterogeneous MPSoCs. In: Proceedings of the 49th annual design automation conference DAC'12. ACM, San Francisco, pp 1266–1271. ISBN: 978-1-4503-1199-1. https://doi.org/10.1145/2228360.2228597

Castrillon J, Leupers R, Ascheid G (2013) MAPS: mapping concurrent dataflow applications to heterogeneous MPSoCs. IEEE Trans Ind Inform 9(1):527–545. ISSN: 1551-3203. https://doi.org/10.1109/TII.2011.2173941

Castrillon J, Lieber M, Klüppelholz S, Völp M, Asmussen N et al (2018) A hardware/software stack for heterogeneous systems. IEEE Trans Multi-Scale Comput Syst 4(3):243–259. ISSN: 2332-7766. https://doi.org/10.1109/TMSCS.2017.2771750. http://ieeexplore.ieee.org/document/8103042/

C/DA Design Automation (2020) IEEE standard for software-hardware interface for multi- many-core. In: IEEE Std 2804-2019, pp 1–84. https://doi.org/10.1109/IEEESTD.2020.8985663. https://standards.ieee.org/standard/28042019.html

Ceng J, Castrillon J, Sheng W, Scharwächter H, Leupers R et al (2008) MAPS: an integrated framework for MPSoC application parallelization. In: Proceedings of the 45th annual design automation conference DAC'08. ACM, Anaheim, pp 754–759. ISBN: 978-1-60558-115-6. https://doi.org/10.1145/1391469.1391663

Ceng J, Sheng W, Castrillon J, Stulova A, Leupers R, Ascheid G, Meyr H (2009) A high-level virtual platform for early MPSoC software development. In: Proceedings of the 7th IEEE/ACM international conference on hardware/software codesign and system synthesis (CODES+ISSS'09). ACM, Grenoble, pp 11–20. ISBN: 978-1-60558-628-1. http://doi.org/10.1145/1629435.1629438

Church A (1985) The calculi of lambda-conversion, vol 6. Princeton University Press, Princeton

Dardaillon M, Marquet K, Risset T, Martin J, Charles H-P (2016) A new compilation flow for software-defined radio applications on hetero-geneous MPSoCs. ACM Trans Archit Code Optim (TACO) 13(2):1–25

de Dinechin BD (2013) Dataflow language compilation for a single chip massively parallel processor. In: 2013 IEEE 6th international workshop on multi-/many-core computing systems (MuCoCoS). IEEE. pp 1–1

de Dinechin BD (2015) Kalray MPPA®: massively parallel processor array: revisiting DSP acceleration with the Kalray MPPA manycore processor. In: 2015 IEEE hot chips 27 symposium (HCS). IEEE, pp 1–27

Dennis JB (1974) First version of a data flow procedure language. In: Robinet B (ed) Programming symposium. Springer, Berlin/Heidelberg, pp 362–376. ISBN: 978-3-540-37819-8

Deroui H, Desnos K, Nezan J-F, Munier-Kordon A (2017) Relaxed subgraph execution model for the throughput evaluation of IBSDF graphs. In: 2017 international conference on embedded computer systems: architectures, modeling, and simulation (SAMOS). IEEE, pp 213–220

Desnos K, Pelcat M, Nezan J-F, Bhattacharyya SS, Aridhi S (2013) Pimm: parameterized and interfaced dataflow meta-model for mpsocs runtime reconfiguration. In: 2013 international conference on embedded computer systems: architectures, modeling, and simulation (SAMOS). IEEE, pp 41–48

Desnos K, Pelcat M, Nezan J-F, Aridhi S (2015) Memory analysis and optimized allocation of dataflow applications on shared-memory MPSoCs. J Sig Process Syst 80(1):19–37

Desnos K, Pelcat M, Nezan J-F, Aridhi S (2016) Distributed memory allocation technique for synchronous dataflow graphs. In: 2016 IEEE international workshop on signal processing systems (SiPS). IEEE, pp 45–50

Dick R (2008) Embedded Systems Synthesis Benchmark Suite (e3s). http://ziyang.eecs.umich.edu/%5C~%7B%7Ddickrp/e3s/

Ecker W, Müller W, Dömer R (2009) Hardware-dependent software. Springer, Dordrecht, pp 1–13

Eker J, Janneck J (2003) CAL language report Technical report, ERL Technical Memo UCB/ERL, Springer Netherlands

Erbas C, Cerav-Erbas S, Pimentel AD (2006) Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design. IEEE Trans Evol Comput 10(3):358–374

Eusse JF, Williams C, Leupers R (2014) CoEx: a novel profiling-based algorithm/architecture co-exploration for ASIP design. ACM Trans Reconfig Technol Syst. https://doi.org/10.1109/ReCoSoC.2013.6581520

Fettweis G, Dörpinghaus M, Castrillon J, Kumar A, Baier C et al (2019) Architecture and advanced electronics pathways towards highly adaptive energy-efficient computing. Proc IEEE 107(1):204–231. ISSN: 0018-9219. https://doi.org/10.1109/JPROC20182874895. https://ieeexplore.ieee.org/document/8565890

Fradet P, Girault A, Krishnaswamy R, Nicollin X, Shafiei A (2018) RDF: Reconfigurable Dataflow (extended version), Technical report INRIA Grenoble-Rhône- Alpes

Gao L, Huang J, Ceng J, Leupers R, Ascheid G, Meyr H (2009) TotalProf: a fast and accurate retargetable source code profiler. In: CODES+ISSS'09: proceedings of the 7th IEEE/ACM international conference on hardware/software codesign and system synthesis. ACM, Grenoble, pp 305–314. ISBN: 978-1-60558-628-1. https://doi.org/10.1145/1629435.1629477

Georgiou K, Kerrison S, Chamski Z, Eder K (2017) Energy transparency for deeply embedded programs. ACM Trans Archit Code Optim (TACO) 14(1):1–26

Gerstlauer A, Haubelt C, Pimentel AD, Stefanov TP, Gajski DD, Teich J (2009) Electronic system-level synthesis methodologies. IEEE Trans Comput-Aided Des Integr Circuits Syst 28(10):1517–1530

Ghasemi A, Cataldo R, Diguet J-P, Martin KJM (2021) On cache limits for dataflow applications and related efficient memory management strategies. In: Workshop on design and architectures for signal and image processing, 14th edn., pp 68–76

Gleim U, Levy M (2013) MTAPI: parallel programming for embedded multicore systems. In: The multicore association

Glover F (1989) Tabu search—part I. ORSA J Comput 1(3):190–206

Goens A, Khasanov R, Castrillon J, Polstra S, Pi-mentel A (2016) Why comparing system-level MPSoC mapping approaches is difficult: a case study. In: Proceedings of the IEEE 10th international symposium on embedded multicore/many-core systems-on-chip (MCSoC-16), Ecole Centrale de Lyon, Lyon, pp 281–288. https://doi.org/10.1109/MCSoC.2016.48

Goens A, Khasanov R, Hähnel M, Smejkal T, Härtig H, Castrillon J (2017a) TETRiS: a multi-application run-time system for predictable execution of static mappings. In: Proceedings of the 20th international workshop on software and compilers for embedded systems (SCOPES'17).

ACM, Sankt Goar, pp 11–20. ISBN: 978-1-4503-5039-6. https://doi.org/10.1145/3078659.3078663

Goens A, Siccha S, Castrillon J (2017b) Symmetry in software synthesis. ACM Trans Archit Code Optim (TACO) 14(2):20:1–20:26. ISSN: 1544-3566. https://doi.org/10.1145/3095747

Goubier T, Sirdey R, Louise S, David V (2011) ΣC: a programming model and language for embedded manycores. In: International conference on algorithms and architectures for parallel processing Springer, pp 385–394

Hascoët J, Desnos K, Nezan J-F, de Dinechin BD (2017) Hierarchical dataflow model for efficient programming of clustered manycore processors. In: 2017 IEEE 28th international conference on application-specific systems, architectures and processors (ASAP). IEEE, pp 137–142

Hautala I, Boutellier J, Nyländen T, Silvén O (2018) Toward efficient execution of RVC-CAL dataflow programs on multicore platforms. J Sig Process Syst 90:1507–1517. ISSN: 1939-8018. https://doi.org/10.1007/s11265018-1339-x

Heulot J, Boutellier J, Pelcat M, Nezan J-F, Aridhi S (2013) Applying the adaptive hybrid flow-shop scheduling method to schedule a 3GPP LTE physical layer algorithm onto many-core digital signal processors. In: 2013 NASA/ESA conference on adaptive hardware and systems (AHS-2013). IEEE, pp 123–129

Heulot J, Pelcat M, Desnos K, Nezan J-F, Aridhi S (2014) Spider: a synchronous parameterized and interfaced dataflow-based RTOS for multicore DSPs. In: 2014 6th European embedded design in education and research conference (EDERC). IEEE, pp 167–171

Heulot J, Pelcat M, Nezan J, Oliva Y, Aridhi S, Bhattacharyya SS (2014) Just-in-time scheduling techniques for multicore signal processing systems. In: 2014 IEEE global conference on signal and information processing (GlobalSIP), pp 25–29. https://doi.org/10.1109/GlobalSIP.2014.7032071

Holmbacka S, Nogues E, Pelcat M, Lafond S, Lilius J (2014) Energy efficiency and performance management of parallel dataflow applications. In: Proceedings of the 2014 conference on design and architectures for signal and image processing. IEEE, pp 1–8

Honorat A, Desnos K, Dardaillon M, Nezan J-F (2020) A fast heuristic to pipeline SDF graphs. In: Orailoglu A, Jung M, Reichenbach M (eds) Embedded computer systems: architectures, modeling, and simulation. Springer International Publishing, Cham, pp 139–151. ISBN: 978-3-030-60939-9

Huang K, Haid W, Bacivarov I, Keller M, Thiele L (2012) Embedding formal performance analysis into the design cycle of MPSoCs for real-time streaming applications. In: ACM Trans Embed Comput Syst (TECS)

Jantsch A (2003) Modeling embedded systems and SoC's: concurrency and time in models of computation. Elsevier, Morgan Kaufmann, San Francisco

Kahn G (1974) The semantics of a simple language for parallel programming. Inf Process 74:471–475

Kahn G, MacQueen D (1976) Coroutines and networks of parallel processes

Kalray Inc (2020) Kalray MPPA3 Coolidge Anouncement. https://www.kalrayinc.com/release-of-third-generation-mppa-processor-coolidge/

Keinert J, Haubelt C, Teich J (2005) Windowed synchronous data flow. Depart Comput Sci 12:28–49

Kelly JL, Lochbaum C, Vyssotsky VA (1961) A block diagram compiler. Bell Syst Tech J 40(3):669–678

Khasanov R, Castrillon J (2020) Energy-efficient runtime resource management for adaptable multi-application mapping. In: Proceedings of the 2020 design, automation and test in Europe conference (DATE). DATE'20. EDA Consortium, Grenoble

Khasanov R, Robledo J, Menard C, Goens A, Castrillon J (2021) Domain-specific hybrid mapping for energy-efficient baseband processing in wireless networks. ACM Trans Embed Comput Syst (TECS), special issue of the 2021 international conference on compilers, architecture, and synthesis of embedded systems (CASES) 20(5s). ISSN: 1539-9087. https://doi.org/10.1145/3476991

Kienhuis B, Deprettere EF, Van der Wolf P, Vissers K (2001) A methodology to design programmable embedded systems. In: International workshop on embedded computer systems. Springer, pp 18–37

Kirkpatrick S, Gelatt CD, Vecchi MP (1983) Optimization by simulated annealing. Science 220(4598):671–680

Klikpo EC, Khatib J, Munier-Kordon A (2016) Modeling multi-periodic simulink systems by synchronous dataflow graphs. In: 2016 IEEE real-time and embedded technology and applications symposium (RTAS). IEEE, pp 1–10

Koliogeorgi K, Voss N, Fytraki S, Xydis S, Gaydadjiev G, Soudris D (2019) Dataflow acceleration of smith-waterman with traceback for high throughput next generation sequencing. In: 2019 29th international conference on field programmable logic and applications (FPL). IEEE, pp 74–80

Kwok Y-K, Ahmad I (1999) Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Comput Surv 31(4):406–471. ISSN: 0360-0300. http://doi.org/10.1145/344588.344618

Lee EA (2006) The problem with threads. Computer 39(5):33–4

Lee EA, Ha S (1989) Scheduling strategies for multiprocessor real-time DSP. In: 1989 IEEE global telecommunications conference and exhibition 'communications technology for the 1990s and beyond'. IEEE, pp 1279–1283

Lee EA, Messerschmitt DG (1987) Synchronous data flow. Proc IEEE 75(9):1235–1245

Lee EA, Messerschmitt DG (1987) Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans Comput 100(1):24–35

Lee EA, Parks TM (1995) Dataflow process networks. Proc IEEE 83(5):773–801

Lee EA, Seshia SA (2016) Introduction to embedded systems: a cyber-physical systems approach. MIT Press, Cambridge, MA

Leroy X (2009) Formal verification of a realistic compiler. Commun ACM 52(7):107–115

Lesparre Y, Munier-Kordon A, Delosme J (2016) Evaluation of synchronous dataflow graph mappings onto distributed memory architectures. In: 2016 Euromicro conference on digital system design (DSD), pp 146–153. https://doi.org/10.1109/DSD.2016.52

Leupers R, Castrillon J (2010) MPSoC programming using the MAPS compiler. In: Proceedings of the design automation conference (ASP-DAC), 2010 15th Asia and South Pacific, pp 897–902. https://doi.org/10.1109/ASPDAC.2010.5419677

Leupers R, Aguilar MA, Eusse JF, Castrillon J, Sheng W (2017) MAPS: a software development environment for embedded multicore applications. In: Ha S, Teich J (eds) Handbook of hardware/software codesign. Springer, Dordrecht, pp 1–33. ISBN: 978-94-017-7358-4. https://doi.org/10.1007/978-94-017-7358-4_2-1

Lin S, Wang L-H, Vosoughi A, Cavallaro JR, Juntti M et al (2015) Parameterized sets of dataflow modes and their application to implementation of cognitive radio systems. J Sig Process Syst 80(1):3–18

Lohstroh M, Romero ÍÍ, Goens A, Derler P, Castrillon J, Lee EA, Sangiovanni-Vincentelli A (2020) Reactors: a deterministic model for composable reactive systems. In: Chamberlain R, Grimheden ME, Taha W (eds) Cyber physical systems. Model-based design – proceedings of the 9th workshop on design, modeling and evaluation of cyber physical systems (CyPhy 2019) and the workshop on embedded and cyber-physical systems education (WESE 2019). Springer International Publishing, New York City, pp 59–85. ISBN: 978-3-030-41131-2. https://doi.org/10.1007/978-3-030-41131-2_4

Madronal D, Arrestier F, Sancho J, Morvan A, Lazcano R et al (2019) Papify: automatic instrumentation and monitoring of dynamic dataflow applications based on papi. IEEE Access 7:111801–111812

Manolache S, Eles P, Peng Z (2008) Task mapping and priority assignment for soft real-time applications under deadline miss ratio constraints. ACM Trans Embed Comput Syst (TECS) 7(2):1–35

Marwedel P, Bacivarov I, Lee C, Teich J, Thiele L et al (2011) Mapping of applications to MPSoCs. In: Proceedings of the 9th international conference on hardware/software codesign and system synthesis (CODES+*ISSS)*,Springer, New York, NY, pp 109–118

Menard C, Goens A, Lohstroh M, Castrillon J (2020) Achieving determinism in adaptive AUTOSAR. In: Proceedings of the 2020 design, automation and test in Europe conference (DATE), DATE'20. EDA Consortium, Grenoble

Menard C, Goens A, Hempel G, Khasanov R, Robledo J, Teweleitt F, Castrillon J (2021) Mocasin—rapid prototyping of rapid prototyping tools: a framework for exploring new approaches in mapping software to heterogeneous multi-cores. In: DroneSE and RAPIDO 2021, system engineering for constrained embedded systems RAPIDO'21. Virtual event. https://doi.org/10.1145/3444950.344728

Mendis C, Renda A, Amarasinghe S, Carbin M (2019) Ithemal: accurate, portable and fast basic block throughput estimation using deep neural networks. In: International conference on machine learning. PMLR, pp 4505–4515

Meneses-Viveros A, Paredes-López M, Hernández-Rubio E, Gitler I (2021) Energy consumption model in multicore architectures with variable frequency. J Supercomput 77:2458–2485

Neuendorffer S, Lee EA (2004) Hierarchical reconfiguration of dataflow models. In: MEMOCODE. https://doi.org/10.1109/MEMCOD.2004.1459852

Orsila H, Kangas T, Salminen E, Hämäläinen TD, Hännikäi-nen M (2007) Automated memory-aware application distribution for multi-processor system-on-chips. J Syst Arch 53(11):795–815

Pelcat M, Menuet P, Aridhi S, Nezan J-F (2009) Scalable compile-time scheduler for multi-core architectures. In: 2009 design, automation & test in Europe conference & exhibition. IEEE, pp 1552–1555

Pelcat M, Nezan JF, Piat J, Croizer J, Aridhi S (2009) A system-level architecture model for rapid prototyping of heterogeneous multicore embedded systems. In: Conference on design and architectures for signal and image processing (DASIP) 2009, Nice, 8pp. https://hal.archives-ouvertes.fr/hal00429397

Pelcat M, Desnos K, Heulot J, Guy C, Nezan J-F, Aridhi S (2014) Preesm: a dataflow-based rapid prototyping framework for simplifying multicore DSP programming. In: 2014 6th European embedded design in education and research conference (EDERC). IEEE, pp 36–40

Pelcat M, Desnos K, Maggiani L, Liu Y, Heulot J, Nezan J, Bhattacharyya SS (2016) Models of architecture: reproducible efficiency evaluation for signal processing systems. In: 2016 IEEE international workshop on signal processing systems (SiPS), pp 121–126. https://doi.org/10.1109/SiPS.2016.29

Pelcat M, Mercat A, Desnos K, Maggiani L, Liu Y et al (2018) Reproducible evaluation of system efficiency with a model of architecture: from theory to practice. IEEE Trans Comput-Aided Des Integr Circuits Syst 37(10):2050–2063. https://doi.org/10.1109/TCAD.2017.2774822

Pellegrini A, Stephens N, Bruce M, Ishii Y, Pusdesris J et al (2020) The arm neoverse N1 platform: building blocks for the next-gen cloud-to-edge in-frastructure SoC. IEEE Micro 40(2):53–62

Piat J, Bhattacharyya SS, Raulet M (2009) Interface-based hierarchy for synchronous data-flow graphs. In: 2009 IEEE workshop on signal processing systems, pp 145–150. https://doi.org/10.1109/SIPS.2009.5336240

Pimentel AD, Erbas C, Polstra S (2006) A systematic approach to exploring embedded system architectures at multiple abstraction levels. IEEE Trans Comput 55(2):99–112. ISSN: 0018-9340. https://doi.org/10.1109/TC.2006.16

Pino JL, Bhattacharyya SS, Lee EA (1996) A hierarchical multiprocessor scheduling system for DSP applications. In: *Conference record of the twenty-ninth asilomar conference on signals, systems and computers*, vol 1. IEEE, pp 122–126

Piscitelli R, Pimentel AD (2011) A high-level power model for mpsoc on FPGA. In: 2011 IEEE international symposium on parallel and distributed processing workshops and Phd forum. IEEE, pp 128–135

Ptolemaeus C (ed) (2014) System design, modeling, and simulation using ptolemy II. Ptolemy.org. http://ptolemy.org/books/Systems

Quan W, Pimentel AD (2015) A hybrid task mapping algorithm for heterogeneous MPSoCs. ACM Trans Embed Comput Syst (TECS) 14(1):14

Rogers P, Fellow A (2013) Heterogeneous system architecture overview. In: Hot chips symposium, pp 1–41

Schuermans S, Leupers R (2019) Power estimation on electronic system level using linear power models. Springer, Cham

Schwarzer T, Weichslgartner A, Glaß M, Wildermann S, Brand P, Teich J (2017) Symmetry-eliminating design space exploration for hybrid application mapping on many-core architectures. IEEE Trans Comput-Aided Des Integr Circuits Syst 37(2):297–310

Sérot J (2020) HoCL: high level specification of dataflow graphs. In: Proceedings of the 32nd international symposium on implementation and application of functional languages (IFL 2020) University of Kent, pp 244–253. https://www.cs.kent.ac.uk/events/2020/ifl20/ifl2020draftproceedings.pdf

Singh AK, Kumar A, Srikanthan T (2011) A hybrid strategy for mapping multiple throughput-constrained applications on MPSoCs. In: 2011 proceedings of the 14th international conference on compilers, architectures and synthesis for embedded systems (CASES). IEEE, pp 175–184

Stemmer R, Vu H-D, Grüttner K, Le Nours S, Nebel W, Pillement S (2020) Towards probabilistic timing analysis for SDFGs on tile based heterogeneous MPSoCs

Stuijk S, Geilen M, Theelen B, Basten T (2011) Scenario-aware dataflow: modeling, analysis and implementation of dynamic applications. In: 2011 international conference on embedded computer systems: architectures, modeling and simulation. IEEE, pp 404–411

Synopsys Signal Processing WorkSystem (SPW) (2013) The Fastest Path from Innovation to Implementation of Digital Signal Processing Systems. http://www.eigen.in/pdf/SPW.pdf

Synopsys System Studio (2010) https://news.synopsys.com/index.php?s=20295&item=123136

Teich J, Henkel J, Herkersdorf A, Schmitt-Landsiedel D, Schröder-Preikschat W, Snelting G (2011) Invasive computing: an overview. In: Multiprocessor system-on-chip. Springer, New York, NY, pp 241–268

The Multicore Association, Inc (2015) Software-hardware interface for multi-many-core (SHIM) specification, V1.0. The Multicore Association, Inc

Thiele L, Chakraborty S, Naedele M (2000) Real-time calculus for scheduling hard real-time systems. In: 2000 IEEE international symposium on circuits and systems (ISCAS), vol 4. IEEE, pp 101–104

Thiele L, Bacivarov I, Haid W, Huang K (2007) Mapping applications to tiled multiprocessor embedded systems. In: ACSD'07: proceedings of the seventh international conference on application of concurrency to system design. IEEE Computer Society, Washington, DC, pp 29–40. ISBN: 0-7695-2902-X. https://doi.org/10.1109/ACSD.2007.53

Tretter A (2018) On efficient data exchange in multicore architectures. PhD thesis. ETH Zurich, 206pp. https://www.research-collection.ethz.ch/handle/20.500.11850/309314

Van Stralen P, Pimentel AD (2010) A high-level microprocessor power modeling technique based on event signatures. J Sig Process Syst 60(2):239–250

Van Stralen P, Pimentel AD (2010) A trace-based scenario database for high-level simulation of multimedia MPSoCs. In: 2010 international conference on embedded computer systems: architectures, modeling and simulation. IEEE, pp 11–19

Weichslgartner A, Wildermann S, Gangadharan D, Glaß M, Teich J (2018) A design-time/run-time application mapping methodology for predictable execution time in MPSoCs. ACM Trans Embed Comput Syst (TECS) 17(5):89

Wilhelm R, Engblom J, Ermedahl A, Holsti N, Thesing S et al (2008) The worst-case execution-time problem—overview of methods and survey of tools. ACM Trans Embed Comput Syst 7(3):1–53. ISSN: 1539-9087. https://doi.org/10.1145/1347375.1347389

Yviquel H, Lorence A, Jerbi K, Cocherel G, Sanchez A, Raulet M (2013) Orcc: Multimedia development made easy. In: Proceedings of the 21st ACM international conference on multimedia MM'13. ACM, Barcelona, pp 863–866. ISBN: 978-1-4503-2404-5. https://doi.org/10.1145/2502081.2502231

Yviquel H, Sanchez A, Mickaël R, Casseau E (2017) Multi-core runtime for dynamic dataflow video decoders, Technical Report. IETR/INSA Rennes, IRISA, Inria Rennes. https://hal.archives-ouvertes.fr/hal-01503378

Retargetable Compilation

32

Gert Goossens, Dirk Lanneer, Johan Van Praet, and Werner Geurts

Contents

Introduction and Historical Perspective 1148
Compiler Construction 1149
Compiler Frameworks 1149
Retargetable Compilers 1151
Outline of This Chapter 1153
Anatomy of a Compiler 1153
Intermediate Representations 1153
Compilation Phases and Dependencies 1154
Architectural Scope of ASIPs 1161
Parallelism 1162
Specialization 1164
Example 1166
Retargetable Compilers for ASIPs 1167
Processor Intermediate Representations 1170
Retargetable Compiler Optimizations 1171
Conclusions 1185
References 1186

Abstract

Compilers are essential tools to implement application software, coded in high-level programming languages, on programmable processors. Compiler construction has been an established engineering discipline for several decades. This has resulted in the availability of common technologies and tool infrastructure for compiler engineers. However, to cope with ever-increasing computational

G. Goossens (✉) · D. Lanneer · J. Van Praet · W. Geurts
Synopsys, Leuven, Belgium
e-mail: gert.goossens@synopsys.com; dirk.lanneer@synopsys.com; johan.van.praet@synopsys.com; werner.geurts@synopsys.com

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_67

demands of electronic system applications, processor architectures have evolved drastically since the early days of compiler engineering. This continues to bring new challenges for compiler developers.

Today's systems-on-chip can contain a significant number of processor cores. Their architectures exhibit higher amounts of parallelism and specialization. An important requirement, driven by new generations of embedded electronic applications, is that processor architectures are tuned for the needs of their application domain, resulting in so-called application-specific processors or ASIPs.

The impact on compilers is multifold. Compilers must be able to exploit the features of highly specialized architectures. For each new architecture, a compiler must become available in very short time. Processor architectures and compilers should be codeveloped so that early compilation of application code can provide feedback to drive architectural decisions.

Retargetable compilers adapt automatically to the architecture of a processor, based on a user-defined processor model. Retargetable compilation received initial attention from compiler researchers in the 1990s, and recently saw renewed interest and adoption as a technology that addresses several of the challenges of today's and tomorrow's embedded electronic applications. This chapter discusses concepts, challenges, and solutions in the domain of retargetable compilation. First, retargetable compilation is positioned in a broader historical perspective of compiler technology development. Next, specific requirements for retargetable compilers are formulated, stemming from the wide architectural scope of contemporary application-specific processors. Finally, models, techniques, and optimization algorithms for retargetable compilers are reviewed.

Keywords

Retargetable compilation · Retargetable code generation · Processor architecture · Embedded processor · Application-specific processor · ASIP · Instruction-level parallelism · Data-level parallelism

Introduction and Historical Perspective

A compiler translates software programs coded in high-level languages like C or C++ into low-level instruction sequences that can be executed on the processor. While long established for general-purpose CPUs in computers and servers, compilers have also become indispensable for embedded processors that are integrated in systems-on-chip (SoCs). Direct coding of instruction sequences in assembly language has become uncommon, and is typically restricted to specific use cases, like boot code, specific device interactions, or occasionally the coding of highly optimized library functions.

Therefore, vendors of chips with embedded processors as well as processor intellectual property (IP) vendors also invested in compiler technology, either through internal developments (Texas Instruments n.d.; NXP Semiconductors n.d.; Arm n.d.; Arm-KEIL n.d.; Synopsys n.d.-a; CEVA 2020; Cadence n.d.-a) and/or through third-party tool vendors (IAR Systems n.d.). Such compilers have been carefully optimized for the efficient utilization of the target processor's resources and instruction set. They are often integrated in a software development kit (SDK) for the target processor that may also contain an instruction-set simulator (ISS), a software and on-chip debugger, and a graphical integrated development environment (IDE).

Compiler Construction

The history of software compilation spans more than half a century. Early compilers were merely translation tools that turned source language statements into small sequences of assembly code. Specific local optimizations were then applied to the assembly code, referred to as peephole optimizations. The translation process would use custom-made translation rules for the specific processor target. Around the early 1980s, *compiler construction* became a more established engineering discipline, focusing on general-purpose CPUs. Central to these developments were the introduction of reusable concepts like intermediate representation (IR) formats and a more common understanding of compilation phases, as well as a basic foundation of optimization algorithms used in those phases (see section "Compilation Phases and Dependencies"). This knowledge accumulated in several reference textbooks on compiler construction. Somewhat iconic was the "Dragon" textbook, nicknamed after its cover image (Aho et al. 1986). Its more recent revision continues to be a basis of many compiler courses today (Aho et al. 2007). Other often-cited compiler textbooks include (Muchnick 1997; Morgan 1998; Fischer and LeBlanc 1991; Allen and Kennedy 2001).

Compiler Frameworks

The development of an optimizing compiler is a significant software project, of which the source code base can turn into millions of lines of code. As soon as compiler construction was established as an engineering discipline, with an interest to support standardized programming languages on multiple processor targets and with shared insights on IRs and optimizations, a need emerged in the research community to develop a shared infrastructure for compiler construction. This development was leveraged by the concept of open-source software that started to gain popularity in the 1980s. The authors refer to such a shared compiler infrastructure as a *compiler framework*.

The introduction of compiler frameworks boosted the productivity of compiler engineering teams, who could focus on the development and tuning of optimization algorithms for their processor targets, while reusing language parsers, data structures and IRs, and commonly applicable optimization functions. While used intensively for general-purpose CPUs in computers, these compiler frameworks also accelerated compiler developments targeted at general-purpose embedded processors integrated in SoCs.

The GNU Compiler Collection or *GCC* (originally: the GNU C Compiler) is one of the first and most successful examples of such a compiler framework (Gough 2004; Griffith 2018). GCC was first released in 1987 by Richard Stallman at Massachusetts Institute of Technology (Stallman 1987), supporting C compilation on VAX and 68K CPUs. By now, GCC supports a dozen programming languages, and GCC-based compilers have been developed for almost 80 different processor targets including embedded processors (Free Software Foundation n.d.).

LCC is a compact compiler framework supporting the C language, originating from Microsoft Research (Fraser and Hanson 2001). It gained popularity in compiler research and teaching. Compiler textbook (Fraser and Hanson 1995) is built around LCC.

LLVM can be considered a second-generation compiler framework that has gained popularity since around 2005 (Lattner and Adve 2004; LLVM Project n.d.-a). While it reuses many of GCC's interface and library formats that became de facto standards, it promises to adapt more easily to both new programming languages and new processor targets through a more modular software architecture.

LLVM's more recent source language front-end *Clang*, although made for C and C++, contains optimizations that are applicable to other programming languages as well (LLVM Project n.d.-b). This has inspired the development of alternative front-ends to LLVM for domain-specific programming languages (see examples in sections "Front End and Middle End"). LLVM also contains a significant set of optimizations in its so-called middle end that are independent of the source language and the processor target. Like with other compiler frameworks, LLVM requires that target-specific compiler back ends (also called code generators) are built. Compiler back ends for existing targets that are upstreamed to the open-source community can be a source of inspiration for such developments.

An important element of LLVM's modular architecture is its pass manager utility, which enables compiler engineers to build compilation flows with alternative passes (i.e., full traversals of all compilation phases) for specific purposes.

LLVM's modular architecture makes it easier to build alternative compilation flows. An example is LLVM's implementation of *link-time optimization* (LTO), which enables code optimizations across the multiple source files that comprise a whole application program (see section "Linker"). Another example is *just-in-time compilation* (JIT), which enables certain code optimizations to be performed at runtime, i.e., while the compiled program is executing.

A recent development in the domain of compiler frameworks is *MLIR*, named after its IR called *multilevel intermediate representation* (see also section "Intermediate Representations"). It provides an infrastructure that further eases the

support of domain-specific programming languages, including language-specific transformations and optimizations. It can directly generate the IR of the LLVM compiler framework, so that LLVM can be used for the remaining compilation phases.

Retargetable Compilers

While compiler frameworks significantly ease the development of a compiler for a given processor target, they still rely on a manual process of entering target-specific instruction patterns and optimizations. Especially the code generator or back end of the compiler requires careful tuning of optimizations. Compiler frameworks assume a certain architectural template for the processor, typically inspired by general-purpose CPU architectures (see section "Architectural Scope of ASIPs"). If the processor target does not fit the template, a significant effort may still be required to tune the available front end and middle end of the compiler framework, and to develop back-end (code generation) phases that are adequate for the processor target.

Since the early 2000s, the electronics industry has delivered strong growth in smart connected products implemented in SoCs. The computational power of these systems is largely delivered by specialized embedded processors such as digital signal processors (*DSPs*) and application-specific instruction-set processors (in short: application-specific processors or *ASIPs*). See the section "Application-Specific Processors" in this volume (for example, ▸ Chap. 7, "Architectures for Multimedia Processing: A Cross-Layer Perspective", "Architectures for Wireless Signal Processing", and ▸ Chap. 10, "Architectures for Machine Learning"). ASIPs offer higher concurrency than general-purpose CPUs, through instruction-level, data-level, and task-level parallelism. Also, they typically use architectural specialization in the supported datatypes, arithmetic and logic operators, storage and interconnect architecture, and instruction pipeline, as needed by the application domain. The performance and power consumption benefits of ASIPs resulted in a proliferation of processor architectures, which in turn called for increased automation in the compiler development process. This resulted in a new concept of retargetable compilers, which is the subject of this chapter.

The authors define a *retargetable compiler* as a compiler that automatically adapts (i.e., retargets) to the architecture of a processor (i.e., the target) based on a processor model (see Fig. 1). The processor model can be modified by the user and read by the tool at any moment, which results in a functional compiler for the modified model without a need for the user to modify the internals (compilation phases and optimizations) of the compiler. The retargetable compiler may either interpret the processor model while compiling application code for the target, or it may offer an automatic process to build a target-specific instance of a compiler from the processor model without user intervention and in limited time. The processor model often takes the form of an architecture description language or *ADL* (also called processor description language). See the companion ▸ Chap. 23, "Architecture Description Languages" for a survey of ADLs.

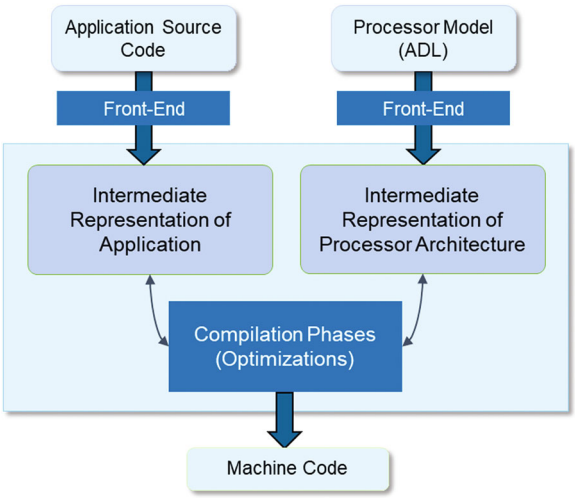


Fig. 1 Conceptual design flow of a retargetable compiler

Retargetable compilers may reuse elements from compiler frameworks like GCC or LLVM. However, the compiler architecture necessarily differs in certain aspects, especially when it comes to the automatic retargeting of the compiler back end (code generator). The authors contend that the term retargetable compiler is sometimes used incorrectly to refer to compiler frameworks.

Retargetable compilation emerged as a dedicated field out of compiler research in the mid-1990s. Relevant research work can be found in an early contributed volume (Marwedel and Goossens 1995), in proceedings of subsequent editions of the International Workshop on Code Generation for Embedded Processors, later renamed into SCOPES (International Workshop on Software and Compilers for Embedded Systems n.d.), and in more recent survey works such as (Leupers and Marwedel 2013).

As of 2000, the development of retargetable compilers based on ADLs spurred the introduction of broader methodologies and tools that encompassed the entire design cycle of ASIP architectures. The idea behind such methodologies is that designers can rapidly explore the performance of an ASIP architecture by describing it in an ADL, use the retargetable compiler to compile representative application benchmarks on the architecture, and measure the performance of the generated code. By profiling the generated code, architectural hotspots can be identified, which reveal opportunities for tuning the ASIP architecture modeled in the ADL, for the intended application domain. Such a rapid architectural exploration cycle is only feasible if retargeting the compiler is an instantaneous process, which is not the case with standard compiler frameworks. A number of commercial ASIP design solutions have become available, such as Synopsys' ASIP Designer tool (Synopsys n.d.-b), Cadence's extensible processor IP product called Xtensa (Cadence n.d.-b), and the Codasip Studio tool (Codasip n.d.-b). In addition to a retargetable compiler, ASIP design tools typically also contain instruction-set simulation and register-

transfer level (RTL) hardware generation tools that work from the same ADL as the retargetable compiler.

The concept of ASIPs and the interest in retargetable ASIP design tools has recently been reinforced by the emergence of RISC-V, an open-source processor architecture technology (Waterman and Asanović 2019; Waterman et al. 2021). One of RISC-V's use models is as a baseline architecture to which designers can add domain-specific extension instructions. The RISC-V instruction-set architecture reserves opcode space to encode such extension instructions. ADL models of RISC-V have been developed, and ASIP design tools are being used to explore instruction extensions and to automatically obtain a compiler, simulator, and RTL implementation of the extended RISC-V processor (Synopsys 2022; Codasip n.d.-a).

Outline of This Chapter

This chapter discusses retargetable compilers for ASIPs. As a basis for the discussion, section "Anatomy of a Compiler" first provides an introduction into the structure of compilers for general-purpose processors, as commonly understood in the engineering community. Section "Architectural Scope of ASIPs" zooms in on the architectural scope of ASIPs, which differs from general-purpose processors in several aspects and therefore imposes specific requirements on retargetable compilers for ASIPs. Section "Retargetable Compilers for ASIPs" then describes how these requirements can be addressed by combining existing and new compiler technologies.

Anatomy of a Compiler

Compilers transform application source code into executable code for the processor target in multiple steps, referred to as *compilation phases*. Within such compilation phases, specific optimization algorithms may be applied. Information about the application code is represented in an internal data structure, called *intermediate representation* (IR). Throughout the compilation process the IR is gradually transformed and refined to reflect successive decisions taken by the compiler. Section "Intermediate Representations" provides a short introduction on IRs. Section "Compilation Phases and Dependencies" discusses optimization phases in more detail.

Intermediate Representations

An IR is an internal data structure used by compilers to represent the application to be compiled. At the start of the compilation process the IR directly represents the application source code. At the end the IR contains all required information to emit

the machine code for the application in assembly or binary form. The IR can reside in computer memory, but many compilers also allow the dumping of the IR in text or binary files (for example, in LLVM's bit-code format) at intermediate points, and the reading of these files to resume the compilation process.

After the initial parsing of the application source code and the initial data-flow analysis (see sections "Front End" and "Middle End") the IR often takes a form that explicitly denotes the following information:

1. Linear code sequences called *basic blocks*, composed of operations and information on their mutual data dependencies. Examples of such representations are:
   - *Static single-assignment* (SSA) form (Cytron et al. 1991). This is essentially a collection of assignment statements in which each assigned variable has a unique name. Two statements have a data dependency when they respectively write and read the same named variable.
   - *Data-flow graphs* (DFGs) (Dennis 2011). These are directed graphs in which nodes represent operations and edges represent data dependencies between them.
2. Control dependencies between the basic blocks. These are typically represented in a directed graph called *control-flow graph*.

For example, the GCC and LLVM compilers' initial phases use SSA forms combined with a control-flow graph. ASIP Designer's retargetable compiler uses a *control and data-flow graph* (CDFG) model, which nests data-flow graphs in a control-flow graph (Van Praet 1997).

More recently, enhanced IRs with support for domain-specific compilation have been proposed. An example is the *multilevel intermediate representation* (MLIR) (Lattner et al. 2021), which uses enhanced data-flow graph models with support for loop structure and memory layout transformations, offered within a compiler framework that supports language-specific transformations and optimizations. MLIR has been used to build compilers for domain-specific programming languages like TensorFlow used for machine learning applications, among others.

Compilation Phases and Dependencies

While the engineering community has established a common understanding of compiler construction and the notion of compilation phases, the granularity and scope of compilation phases, their dependencies, and the optimizations performed in them may differ depending on the design decisions in each compiler, and also depend on the terminology used by its authors.

Figure 2 shows a commonly accepted structure of a compiler. It consists of four main stages (see left-hand side of the figure): a source language front end, a middle end or optimizer, a back end or code generator, and a linker (which may also be considered a tool separate from the compiler). The right-hand side of the figure lists typical examples of processing and optimization steps within those stages. Some

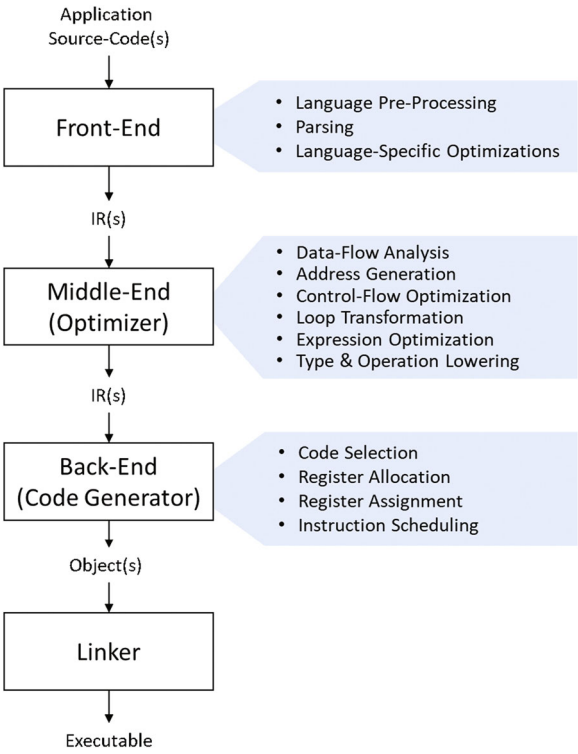


Fig. 2 Typical stages in a compiler (left) and examples of compilation phases (right)

authors would refer to the four stages (left hand) as compilation phases, others would consider the more detailed steps (right hand) as compilation phases. In this chapter, the latter convention is followed.

Compilation phases are often executed in a predetermined sequence. However, the compiler may apply mechanisms for *phase coupling,* i.e., the fact that mutual dependencies between phases must be accounted for to generate efficient executable code. This may be achieved by using predictors for a late phase's optimization result in an earlier phase, by backtracking from a late to an earlier compilation phase, or by applying certain phases multiple times in a full compilation pass.

Front End

The compiler front end parses all application source files and builds an IR representation for them. Parsing consists of lexical, syntax, and semantic analysis steps (Grune and Jacobs 1990). Well-known parser generation tools include Lex/Yacc (Levine et al. 1992), Flex/Bison (Levine 2009), and ANTLR (Parr 2007). Prior to actual parsing, a language preprocessor may be called. This is essentially a language-independent tool that can perform tasks like conditional activation of code portions, header file inclusion, and macro expansion.

Middle End

The compiler middle end performs optimizations on the IR that are in principle applicable regardless of the processor target. In reality, some optimizations may only make sense if the processor target contains certain instructions or hardware features, and thus may be omitted if this is not the case. The optimizations are implemented in distinct compilation phases. Some of the most common compilation phases are described next.

Data-Flow Analysis

This initial phase builds the SSA form or data-flow graph and the control-flow graph, which represent the application source code. An important part of the analysis is to determine the valid data dependencies between operations that define named variables and operations that use named variables. The input is an IR description that represents a sequential order of statements (assuming that the source code is described in a sequential language like C or C++). For every assignment and every use of a given variable, also a *reaching definitions* analysis is carried out, to determine if the variable will not intermediately be overwritten by another assignment. If a variable is assigned multiple times, each assignment gets a unique variable instance name, and all uses of the variable are replaced by uses of the right instance. Data-flow analysis is important to enable the exploitation of instruction-level parallelism (see instruction scheduling phase in the compiler back end, in section "Back End").

Alias analysis (also known as *points-to analysis* or *memory reference disambiguation*) is another important part of data-flow analysis. It deals with memory references in the application source code in the form of global or static variables or of pointers. It checks whether two memory references can ever refer to the same memory location, in which case a dependency must be assumed if at least one of the references is a write-to-memory operation. An example of alias analysis (explained using C code) is shown in Fig. 3.

Address Generation

In this phase, address expressions are introduced for global and static variables in the application code that are to be stored in memories, considering the addressing modes that are available on the processor target. The computed results of these address

```
int A[8], B[8];

A[i] = 10;            // Update of A
A[j] = 11;            // Next update of A: depends on previous

B[j] = 20;            // Update of B: independent of A updates

int* p = c ? A : B;

*p = 30;              // Update of A[0] or B[0]: depends on all
                      // previous A and B updates
```

Fig. 3 Example of alias analysis

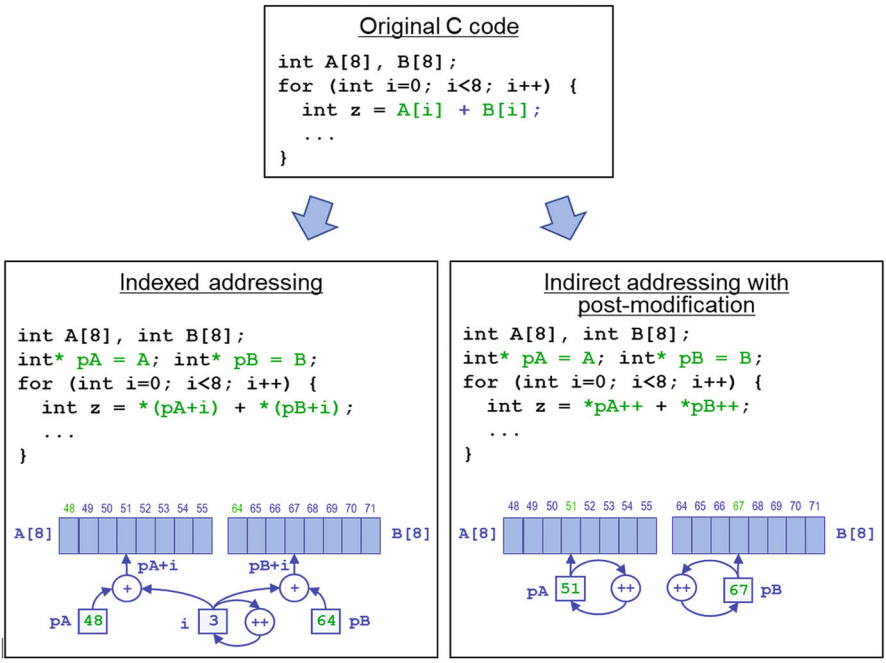


Fig. 4 Pointer analysis, assuming that the processor supports indexed addressing (bottom left) or indirect addressing wtth pointer post-modification (bottom right). In the latter case, arrays **A[]** and **B[]** can be accessed with independent pointers that are induction variables. Note: This example assumes a word-addressable memory, with **int** mapping into a single word

expressions will serve as inputs to load or store instructions or arithmetic or logic instructions that use memory operands.

When the application code contains global or static array references, the arrays' index expressions are translated into pointer arithmetic. An optimization that can be applied here is *induction-variable analysis*. An induction variable is a variable that is increased or decreased by a fixed amount on every iteration of a loop, or that is a linear function of other induction variables. Induction-variable analysis is illustrated in Fig. 4. The original C code shown at the top contains array references with index expressions. At the bottom left a translation into pointer arithmetic is shown suited for a processor with indexed addressing modes. At the bottom right a more efficient translation is shown in case the processor supports indirect addressing with post-modification of address pointers. Arrays **A[]** and **B[]** can then be accessed with independent pointers that are induction variables. These address computations are less complex and offer more freedom to the instruction scheduler in the compiler back end, as illustrated by the data-flow graphs below the code. Moreover, indirect addressing typically results in better timing performance (i.e., higher clock frequencies) than indexed addressing, since it avoids aritmetic operations in the path to the address bus.

Control-Flow Optimization
This phase deals with control-flow statements in the source code, represented in the application's control-flow graph. Multiple optimizations can be applied, aiming at the generation of faster or more compact machine code. A few examples are discussed next.

Function in-lining is a process whereby function calls are substituted by the instantiated body of the function that was called. This avoids the overhead associated with the subroutine call and return mechanisms that would traditionally be used to implement function calls, at the expense of increased code size. After in-lining, each instance of the function will be optimized specifically within its own context. C and C++ compilers can be given hints to perform function in-lining by annotating function calls with the **inline** specifier in the application source code. Many compilers use heuristics to decide about the automatic in-lining of functions.

Another control-flow optimization relates to the implementation of *switch statements*. Small switch statements can be implemented as conditional branches, but for larger ones the compiler may introduce an array of pointers to different parts of the code in program memory, called a *jump table*.

A final control-flow optimization example is the replacement of a jump-based implementation of small-size *if-then-else* blocks of code by either *speculative* or *predicated execution*. In case of speculative execution, the operations in both the *then* and the *else* branch are executed unconditionally followed by a conditional selection of the results. This requires that the processor target has a conditional select instruction. In case of predicated execution, the operations in both branches are executed but they are guarded by opposite conditions. This requires that the processor target has guarded instructions, i.e., instructions with an additional Boolean input that determines if the results will be actually stored.

Loop Transformation
Loops (such as *for*-loops and *while*-loops) are control-flow statements in the source-code language that specify iteration. By default, a loop can be implemented with a conditional branch instruction that jumps back to the entry point as long as the conditional loop test evaluates true. In the loop transformation phase, the compiler may decide to implement loops in more efficient ways. Some processors offer *zero-overhead loops*, whereby a loop test based on a loop counter is executed transparently in hardware while the instructions from the loop body are executing. The compiler determines which *for*-loops from the application can be implemented using zero-overhead loops. Only a limited number of nested *for*-loops can be mapped into zero-overhead loops, due to capacity limitations of the loop test hardware.

Certain processors support *single-instruction multiple-data (SIMD)* processing, a form of data-level parallelism with instructions operating on vector datatypes. In such cases, the loop transformation phase in the compiler may try to introduce auto-vectorization, i.e., transform loops with scalar code into vector code. Auto-vectorization has roots in early Fortran compilers (Allen and Kennedy 1987), is supported for sub-word parallelism (also known as packed SIMD) in compilers for

CPUs like Intel MMX (Bik et al. 2002), and is being researched for wider use cases, for example, in LLVM (LLVM Project n.d.-c).

Expression Optimization
Multiple optimizations can be applied to arithmetic and logic expressions represented in the IR, with the aim to generate faster or more compact machine code. A few examples are discussed next.

- *Strength reduction* replaces operations with equivalent but less expensive ones. For example, in case of array index expressions in loops, multiplications can often be replaced by additions.
- *Constant folding* is the process of recognizing and evaluating constant expressions at compile time rather than computing them at runtime. Terms in constant expressions are typically simple literals, such as the integer literal 2, but they may also be variables of which the value is known at compile time.
- *Common sub-expression elimination* is an optimization that searches for instances of identical expressions (i.e., they all evaluate to the same value) and analyzes whether it is worthwhile replacing them with a single variable holding the computed value.
- *Dead-code elimination* removes expressions from the program of which the result is never used.

Type and Operation Lowering
The application source code may contain datatypes and operators or functions that cannot be mapped directly onto datatypes and functions that are physically supported by the processor target. The latter are referred to as *primitive datatypes* and *primitive functions*. The compiler then must expand the variables of such source-code types into (typically multiple) variables of primitive types, and such source-code operations into (typically multiple) primitive functions. This is referred to as *lowering*.

As an example, assume that C's **long long** data type has a size of 64 bits (i.e., the minimal size required by the language standard). On a 32-bit processor target, variables of type **long long** will then be lowered into pairs of two 32-bit variables, and an addition operator on two **long long** variables will be lowered into a pair of 32-bit additions with an intermediate transfer of the carry bit.

Back End

The back end of the compiler, also called the code generator, is the stage in the compiler where the optimized IR is mapped into a sequence of machine instructions for the target processor. As mentioned before, compiler frameworks typically assume that target-specific back ends are built, which may reuse some common infrastructure or optimizations. In the context of ASIP design tools, the concepts and technologies used in the back end had to be reconsidered, to enable fast automatic retargetability of the back end within a wide scope of ASIP architectures. This will be addressed in sections "Architectural Scope of ASIPs" and "Retargetable

Compilers for ASIPs". Nonetheless, the definition of compilation phases in the back end as known from compiler construction still applies.

For the purposes of this chapter, the following compilation phases are distinguished (details are described in sections "Code Selection" to "Instruction Scheduling"):

- *Code selection*. This phase partitions the IR into small patterns of operations, referred to as *operation bundles*, that can each be implemented in a single instruction of the processor target. Typically, the operations in one such pattern are connected by data dependencies.
- *Register allocation*. This phase allocates the variables that constitute inputs or outputs of operation bundles to storage locations in the processor, i.e., to registers or data memories.
- *Register assignment*. This is sometimes considered part of register allocation. Most processors have register files, i.e., groups of registers with common read and write ports of which the individual register fields can be directly addressed from the instruction word. Register assignment then refers to the selection of individual registers fields for variables allocated to the same register file.
- *Instruction scheduling*. This phase orders the execution of the operation bundles in time. The objective typically is to determine an instruction schedule that minimizes the number of instruction cycles required to execute the application. To that effect, the scheduler must exploit the instruction-level parallelism that is supported by the processor. On the one hand, the scheduler will try to overlap the execution of consecutive instructions by applying instruction pipelining. On the other hand, if the processor supports instruction-word parallelism the scheduler will try to schedule multiple operation bundles in parallel, merging them into a single parallel instruction.

Linker

Once all source files in a program have been compiled to object files, the linker combines these object files into a single executable file. This may also include any precompiled library functions that were called from the application source code. An important function of the linker is to substitute all references to data and program memories that occur in the object files, with absolute addresses in these memories. This is called *relocation*.

Since the linker has a view on the code from the complete program, there is a potential for performing so-called *whole-program optimizations* in the linker. For example, the compiler's middle end may already have performed function in-lining on the code that stems from a single source file. At link time, the linker may perform additional in-lining of functions defined in one source file at the point where they are called in another source file. It is however not obvious for the linker to undo sophisticated optimizations that were already applied at file level in the compiler's middle end and back end. Compilers like GCC and LLVM offer

a *link-time optimization* option, which delays such optimizations to the linker. The linker is given access to higher-level IRs for code from multiple source files. It then combines them into a single IR representing the whole program, and calls the middle end again to apply its optimizations.

Linkers sometimes also perform local optimizations to reduce the code size (i.e., number of bytes in program memory) of the eventual executable. *Reverse in-lining* is an optimization that searches for instruction sequences in the code that occur multiple times in the exact same form, and replaces each occurrence of such a sequence by a call to a subroutine that implements the sequence once. This slightly increases the application's cycle count due to the subroutine call and return overhead.

Architectural Scope of ASIPs

Many models and techniques for compiler construction that have been developed over time and found their way into popular compiler frameworks like GCC and LLVM (see section "Anatomy of a Compiler") were originally intended for general-purpose CPUs. Typical architectural characteristics of general-purpose CPUs include:

- They have a single central register file of which all the register fields are equally accessible as sources or destinations of arithmetic and logic instructions.
- The allowed sizes (i.e., number of bits) of data types are restricted to powers of 2.
- There is (only) a single address space for memories.
- Memories are (only) byte addressable.
- There is no distinction between memory address (i.e., pointer) and integer datatypes.
- Instruction-word parallelism is restricted. CPUs often (only) support the sequential execution of instructions that do not control parallel functional units, albeit that some CPUs may support dynamic multi-issuing of such instruction sequences onto a limited number of parallel units.

It can be noted that GCC and LLVM have been ported to processor architectures with different characteristics, but only at the expense of significant custom development.

While ASIP architectures often reuse a number of architectural features from general-purpose CPU architectures, they typically differ in many respects. Figure 5 lists various features that, in the authors' view, can be found in the instruction-set architecture (ISA) of contemporary ASIP architectures.

Two dimensions are distinguished in the optimization space: *parallelism* and *specialization.* ASIPs will typically combine selected elements from both dimensions, yielding superior performance for the targeted application domain.

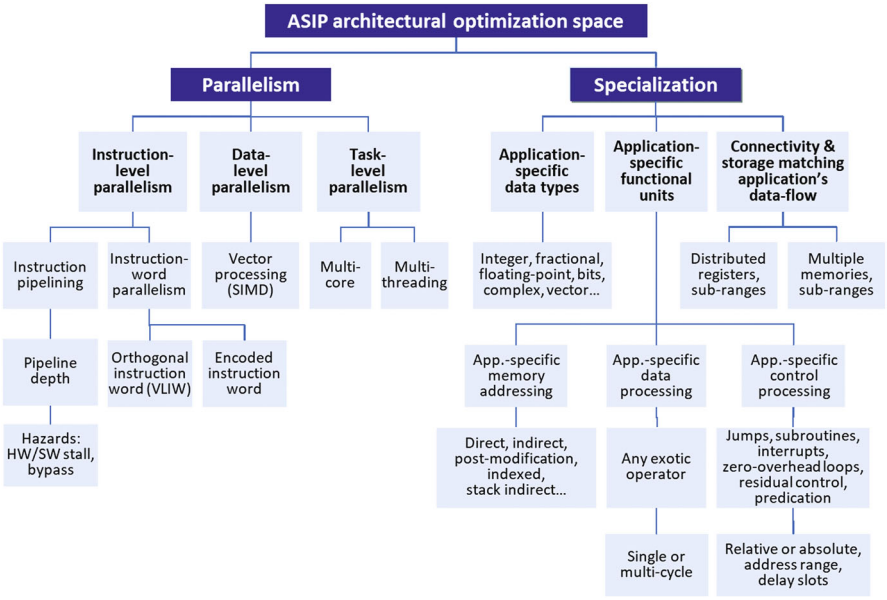


Fig. 5 Instruction-set architectural optimization space for ASIPs

Parallelism

ASIPs can combine three forms of parallelism: *instruction-level parallelism*, *data-level parallelism*, and *task-level parallelism*.

Instruction-Level Parallelism

Instruction-level parallelism (ILP) can be achieved by combining instruction pipelining and instruction-word parallelism.

(a) *Instruction pipelining*

The instruction pipeline of an ASIP can have a custom number of stages that best matches the complexity of the logic design with the timing characteristics of the chosen process technology and cell libraries. In practice, pipelines of ASIPs tend to be shallower than of general-purpose CPUs, because the target application performance is obtained by applying instruction-word parallelism and specialization (see below) while keeping the clock frequency low. This results in lower power consumption and more predictable performance due to the reduced branch delays. Typical pipeline depths of contemporary ASIPs are on the order of 3–6 stages: instruction fetch, instruction decode, up to three execution stages, and optionally a write-back stage.

ASIPs can use different mechanisms to resolve pipeline hazards. Some designs would use a protected pipeline, in which the hardware inserts stall

cycles (also called interlocks) to resolve hazards. More frequently, ASIPs use an exposed pipeline, where the compiler must schedule instructions in such a way as to avoid hazards, if needed by inserting software stalls (i.e., *no-operation* or *nop* instructions). This has the advantage of better cycle-time predictability at compile time. Additionally, a common mechanism to eliminate hazards due to data dependencies in ASIPs is to add *bypass networks* for register files, which can directly feed computational results of one instruction to the inputs of a subsequent instruction before the results are actually stored in the register file.

(b) *Instruction-word parallelism*
This implies that multiple operations can be executed concurrently on different functional units, controlled from a single instruction in the program. These parallel instructions are selected and scheduled statically, i.e., at compile time, by the compiler.

The instruction word can be *orthogonal*, meaning that it is composed of multiple fields (also called *slots*) that each controls one such functional unit. This typically results in a *very-long instruction-word (VLIW)* architecture. ASIPs however often support instruction-word parallelism in *encoded instruction words*. In this case, only those combinations of parallel operations are supported that are considered useful for the application domain. The supported combinations can be encoded in a shorter instruction word, which saves space in program memory and reduces power that would otherwise be consumed in large instruction fetches. ASIPs may also have *variable-length instructions*, with short instructions encoding mostly single operations and longer instructions encoding parallel operations and immediate values (i.e., constants loaded from program memory).

Note that the concept of *dynamic multi-issuing,* whereby the processor hardware tries to execute instructions from a sequential stream in parallel (as in *superscalar* processors), is less common in ASIP architectures. Multi-issuing results in less predictable performance, which is undesirable as ASIP applications typically have real-time performance constraints.

Data-Level Parallelism

This form of parallelism is useful for application domains with large datasets in which identical operations must be applied to multiple data items. Examples include pixels in an image or a video frame, or subcarriers in an OFDM wireless modem. By organizing the data in vectors, and letting single instructions apply the same operation concurrently to all vector elements, high performance can be reached while keeping the instruction word short. This is referred to as *vector processing* or *single-instruction multiple-data (SIMD) processing*.

Some architectures may introduce sub-word parallelism by mapping vectors with narrow elements onto regular data words, such as four 8-bit elements onto a 32-bit word. This is referred to as *packed SIMD*. Most ASIPs with SIMD support however contain separate functional units, registers, and memories that support significantly wider vectors. Today the term *wide SIMD* is often used to refer to vector sizes on the order of 1Kbits or higher. ASIPs often support multiple combinations of vector size

and element word length mapped on the same processor resources. For example, a 512-bit data path may support both vectors with 32 elements of 16 bits each and vectors with 16 elements of 32 bits each.

Task-Level Parallelism
Whereas ILP and SIMD aim at exploiting parallelism in a single thread of control, task-level parallelism refers to the parallel execution of multiple threads of control (i.e., independently evolving program parts or tasks). One such solution is a multicore architecture, in which every core is an ASIP that may be optimized for its tasks. An alternative solution is a single ASIP that supports multithreading. In this case, multiple tasks are interleaved using the same functional units. Typically, but not necessarily, each task uses its own set of registers.

Specialization

Architectural resources of ASIPs can be specialized to the needs of the application domain, as described next.

Datatypes
In addition to the built-in datatypes offered by the application programming language, ASIP architects can introduce any application-specific datatypes. They can be *primitive datatypes* that are physically supported by the processor resources, or else they have to be expanded into primitive datatypes during the lowering phase in the compiler's middle end (see section "Middle End").

Typical examples of datatypes on ASIPs include integer, fractional, floating-point, bit strings, complex, and vector (SIMD) types. These datatypes are not restricted to sizes that are powers of two but can have any number of bits that best suit the application domain. To reduce power consumption and silicon area, the ASIP architect may reduce datatype sizes to the minimum number that still supports the dynamic range required by the applications. For example, convolutional neural network applications often require less than 8-bit precision to represent intermediate-layer data without affecting the classification accuracy (Moons et al. 2016). On the other side of the spectrum, high-performance SIMD architectures may have resources carrying vectors of hundreds of bits (see section "Parallelism").

The interpretation of the bits can be customized. For example, different from the IEEE 754 standard for floating-point arithmetic, one may define a floating-point datatype with custom sizes for mantissa and exponent, or without support for special numbers and exceptions if those are not required by the application. ASIPs for machine learning applications may support the *bfloat16* format (Wang and Kanwar 2019).

Functional Units
Functional units are the resources that implement the ASIP's primitive functions. ASIPs often have an arithmetic and logic functional unit (ALU), with primitive

functions that directly implement built-in operators of the application programming language. ASIPs typically also have application-specific functional units, with primitive functions that implement larger operation patterns in a single instruction that would require multiple instructions on a general-purpose processor. Basic examples include multiply-accumulate (MAC), shift-round-saturate, or FFT butterfly operations, but in fact any combinational logic function with any number of inputs and outputs can be defined as a primitive function. If required for timing reasons, a primitive function can be internally pipelined, resulting in a multicycle primitive function. In special cases, for example, a divide or square-root function, a primitive function can internally reuse resources controlled by a small finite-state machine. In these examples, the number of cycles to completion can be data dependent.

Primitive functions for which no corresponding built-in operator exists in the programming language are often invoked from the application program through so-called *intrinsic function* calls. An intrinsic function is a function in the source code that is directly mapped by the compiler into an instruction that executes the primitive function. This contrasts with regular functions, which are implemented as subroutines (using call and return instructions).

ASIPs often have dedicated functional units to compute memory addresses, called *address generation units (AGUs).* Such AGUs may implement primitive functions that accelerate the execution of address expressions for the ASIP's memory addressing modes. Common addressing modes on ASIPs include direct addressing, indirect addressing with or without post-modification, and indexed addressing with respect to a stack pointer.

Program Control Unit (PCU)

The PCU is a special-purpose functional unit that computes the next program counter (PC) value in each instruction cycle. Similar to general-purpose CPUs, the PCU of any ASIP will support unconditional and conditional branches, as well as subroutine calls and returns. Most ASIPs support zero-overhead loops. The PCU then contains a stack of loop control register triplets, which store the loop start and end addresses in program memory, and the remaining loop count. Target addresses of all control instructions can be made absolute or relative, depending on the needs.

ASIPs may also support several advanced mechanisms for program control. *Residual control* is a mechanism to make the behavior of primitive functions depend on control bits set in mode registers. *Predicated execution* refers to the guarded execution of instructions (see control-flow optimization in section "Middle End"). Vector *predication* is a special form of predicated execution on SIMD architectures. In this case, predicated execution is independently applied to set the value of each vector element, using a vector of conditional guard bits.

Connectivity and Storage Architecture

Many ASIPs have a *load/store* architecture, which means that functional units read data from and write results to registers, and *load* and *store* instructions are available to move data between memories and registers. The latter are typically executing in

parallel with the operations in the functional units. However, in some cases, direct memory operands may be preferred.

The storage architecture (i.e., registers and local data memories) of an ASIP and the connectivity between functional units and storages are often chosen such that they mimic typical data-flow patterns from the application code. Such ASIP architectures often have multiple specialized data memories and small distributed register files or individual registers that are locally connected to inputs and outputs of functional units. This is referred to as a *heterogeneous storage* architecture. As a result, these data-flow patterns can be mapped to the architecture with minimal overhead in terms of the required data moves, resulting in highly efficient code.

As mentioned in section "Parallelism", there is often a desire to keep instruction words relatively short, in order to save space in program memory and reduce power consumption. Distributed, locally connected register files help to reduce the instruction word length, as they only require few opcode bits. In ASIPs with high amounts of instruction-word parallelism, further reductions can be obtained by only allowing sub-ranges of register files as operand or result registers, and by introducing register coupling. Examples of the latter are instructions for which the destination register is always equal to one of the operand registers (referred to as a *read-modify-write* instruction), and instructions that always use the same index for their left and right operand register files.

Example

Figure 6 shows an example ASIP architecture optimized for FFT and DFT algorithms in wireless baseband applications, taken from (Brockmeyer 2010; Goossens 2021). It illustrates several of the ASIP architectural features introduced in

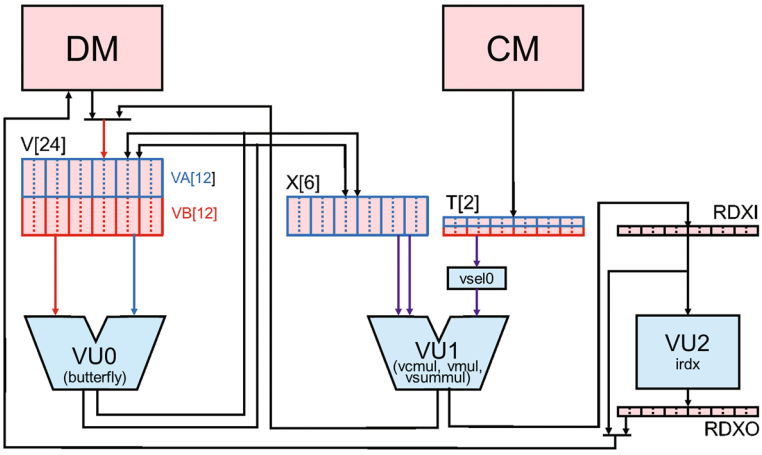


Fig. 6 Functional units, storages, and interconnect architecture of an example ASIP optimized for FFT and DFT, illustrating multiple forms of parallelism and specialization

sections "Parallelism" and "Specialization". The ASIP is optimized for the efficient implementation of the Good-Thomas prime-factor algorithm for DFT.

The example ASIP is a SIMD architecture with a vector size of 192 bits, composed of 6 elements of 32 bits. Each element represents a complex number with 16 bits for both the real and the imaginary part. The choice of 6 elements stems from the fact that all DFT sizes that must be supported in the prime-factor algorithm are multiples of 6. By exploiting this fact, a smaller silicon area and power consumption can be obtained than what is possible with general-purpose vector processors.

The ASIP has three specialized functional units. **VU0** implements special primitive functions for butterfly operations. **VU1** implements basic vector multiplications but also more complex operation patterns like **vsummul()**, which adds the results of two complex vector multiplications and occurs frequently in the application code. The vector multiplier is a two-cycle pipelined multiplier. **VU2** is a dedicated functional unit for implementing special radix computations.

The ASIP has two separate vector memories that can be accessed in parallel: **DM** for data and **CM** for coefficients. Each memory comes with its own AGU (not shown in the figure), supporting indirect addressing modes with pointer post-modifications.

The ASIP's heterogeneous storage architecture is visible in the figure. Multiple small register files of different sizes are locally connected to inputs and outputs of functional units. Register files **V[]** and **T[]** are both partitioned in a lower and a higher sub-range with half the number of fields. While some instructions can access the full register file, others can only access sub-ranges (as indicated by black, red, and blue colors in the figure).

The ASIP supports instruction-word parallelism in up to five parallel slots: three slots dedicated to each of the functional units, and two slots dedicated to memory loads and stores. The ASIP has variable-length instructions, shown in Fig. 7. A 64-bit instruction format is used to encode instructions with five slots. A shorter 32-bit format is used to encode stand-alone (i.e., nonparallel) instructions for control, memory loads and stores, and vector operations. Finally, a 16-bit format is used for stand-alone scalar RISC instructions.

The ASIP has a four-stage instruction pipeline, composed of an instruction fetch, an instruction decode, and two execution stages.

Retargetable Compilers for ASIPs

As defined in section "Introduction and Historical Perspective", a retargetable compiler automatically adapts to the processor target based on a processor model. Retargetable compiler technology is essential in the context of ASIP designs, for several reasons.

First, due to their application-specific nature, every ASIP architecture can be expected to be different. The manual design or porting of an optimizing compiler for each ASIP instance, even when using compiler frameworks, is too costly and time consuming. Second, architectural exploration to determine the best ASIP architecture for the intended application domain is best done by using a retargetable

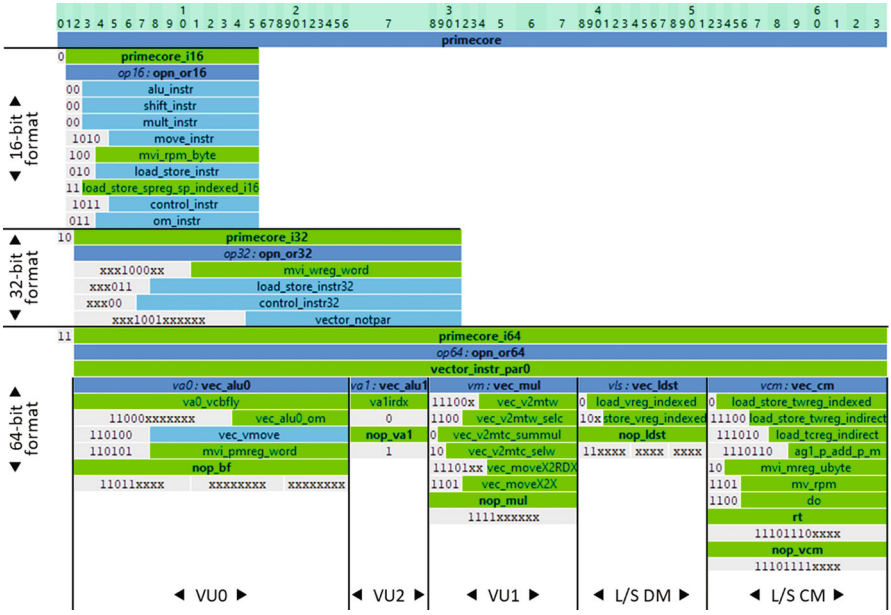


Fig. 7 Instruction formats of the example ASIP for FFT and DFT

compiler, to compile representative application benchmarks, profile the generated code and measure its performance, and use the resulting insights to tune the ASIP architecture.

The ASIP context imposes the following requirements on the compiler:

- It must be fully (i.e., automatically) retargetable.
- It must support a broad architectural scope, as described in section "Architectural Scope of ASIPs".

While a retargetable compiler for ASIPs may reuse elements from established compiler frameworks, its architecture and technology will differ in multiple respects:

- A retargetable compiler reads a processor model defined in an ADL that captures the complete ISA definition and at least those aspects of the microarchitecture that are relevant for the compiler. From this ADL, a front-end tool must build an intermediate representation that represents the architecture, that is easily accessible by all compiler phases, and that offers an efficient data structure for implementing optimizations. One can refer to such an intermediate representation as a *processor IR*. This representation is complementary to the more conventional IR that represents the application code as discussed in section "Intermediate Representations". The task of the retargetable compiler can then

be viewed as mapping the (application) IR onto the processor IR. Processor IRs are discussed below in section "Processor Intermediate Representations".

- Optimization algorithms must be available that directly work on the processor IR, and thus are applicable to any architecture that is described therein, covering the full scope of ASIP architectures, and that result in highly efficient machine code. This mostly pertains to the back end of the compiler, which in traditional compilers is often custom-made for the processor target. Optimization algorithms in retargetable compilers are discussed below in section "Retargetable Compiler Optimizations".

As an example, Fig. 8 shows the internal architecture of the retargetable compiler in the ASIP Designer tool (Goossens 2021). It uses a CDFG representation for the (application) IR, and a so-called instruction-set graph (ISG) representation for the

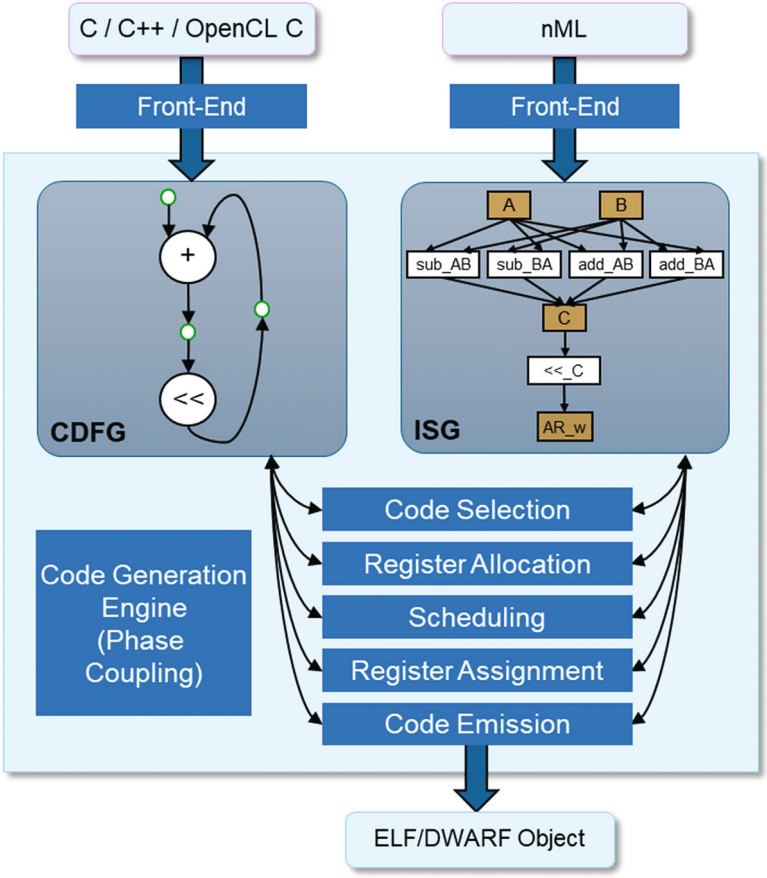


Fig. 8 Internal architecture of ASIP Designer's retargetable compiler

processor IR that is automatically built from processor descriptions expressed in the *nML* ADL (Van Praet et al. 2008).

Processor Intermediate Representations

Compiler frameworks like GCC and LLVM provide internal data structures that can be refined and tuned by compiler engineers to capture information about the specific processor target.

In case of LLVM, these data structures are C++ classes that can be specialized per processor target, referred to as *target description classes* (LLVM Project n.d.-d). The main class **TargetMachine** provides virtual methods to access processor-specific information from more specific classes that contain information on parts of the processor. Examples of such specific classes include:

- **TargetInstrInfo**: information on the available instructions, with their number of operands, opcode mnemonics, etc.
- **TargetLowering**: enumerates supported patterns of control-flow and data-flow graph operations from the application IR and their mapping onto the available instructions. These patterns take the form of directed acyclic graphs (DAGs).
- **TargetRegisterInfo**: information on all registers, grouped in a single register file.
- **DataLayout**: information on memory utilization, such as memory layout, pointer size, and alignment constraints in memory.
- **TargetFrameLowering**: information on the supported layout of a software stack.

As illustrated by the LLVM example above, but the same holds for GCC, the processor IR in established compiler frameworks consists of a collection of multiple representations, each geared at a different phase in the back end of the compiler. By splitting information over multiple representations, some redundancy is introduced which bears a risk of inconsistencies and thus may require extra design and verification effort. Also, such a split poses some challenges in coping with phase coupling efficiently. Besides this, the information in LLVM's processor IP is somewhat geared to general-purpose CPU architectures, which requires a customization effort to support alternative architectures such as ASIPs.

While the information in the processor IR of LLVM or GCC can be understood and entered by compiler engineers, this is less obvious to be accomplished by architecture designers who would benefit from retargetable compilation for architectural exploration. Also, it is not straightforward to automatically generate a processor IR in this format from an ADL that describes complete ASIP architectures from the ground up. For ASIP architectures that are based on a predefined processor template to which extension instructions can be added using an ADL, such an approach is feasible though. An example of the latter is the Xtensa C compiler, which is based on LLVM (Cadence n.d.-b), with automatic support of extension instructions defined in the *TIE* ADL (Sanghavi and Andrews 2008).

Early research in retargetable compilers already explored alternative processor IRs with the intention to generate them from an ADL. Among the more successful

approaches were those aiming at capturing the processor description in a structural model, in the form of a graph. The nodes in such a graph essentially represent hardware resources of the processor, and edges represent their connectivity. The nodes are annotated with behavioral information about the instruction set.

An early academic example of such a graph-based processor IR is the connection-operation (CO) graph model, used in the MSSQ (Nowak 1987) and Record (Leupers and Marwedel 1998) compilers based on the *Mimola* ADL.

A more recent example is the *instruction-set graph* (ISG) (Van Praet 1997; Van Praet et al. 2001) used by the retargetable compiler in the ASIP Designer tool, which can be automatically constructed from processor descriptions in the *nML* ADL. An ISG is a directed bipartite graph, in which the nodes alternatingly represent storage elements and operations, and edges represent the connectivity of the processor. A portion of an ISG example is shown in Fig. 9:

- Nodes representing storage elements are depicted as boxes with a black boundary. They are labeled with the name of the storage, followed by its primitive data type (between brackets). A distinction is made between "static" storages (nodes with blue labels), which hold data until explicitly overwritten, and "transitory" storages (nodes with black labels), which only hold data for a fixed time. Static storages include controllable registers and memories. Transitory storages include pipeline registers that have one cycle delay and wires (nets) that have zero delay.
- Nodes representing operations are depicted as colored boxes. The green boxes represent primitive functions, and the red boxes represent data move operations. They are labeled with a name and annotated with enabling conditions. The latter are compact representations of the different binary encodings of the instruction word that can enable the operation.
- Directed edges representing the connectivity are depicted in black. Combined, these edges indicate how data can flow from storage, through operations, to storage.

An ISG can be interpreted as a superposition of all legal data-flow and control-flow patterns that can be executed on the processor. Several optimization algorithms in the back end of ASIP Designer's compiler have been implemented using graph-theory algorithms that directly operate on the ISG (see section "Retargetable Compiler Optimizations"). Alternatively, an ISG can also be interpreted as a high-level netlist of the processor. In addition to being an input for a retargetable compiler, the ISG model therefore also serves as an input for other tool components in ASIP Designer, such as an instruction-set simulator (ISS) generator and a synthesizable register-transfer level hardware generator.

Retargetable Compiler Optimizations

In this section, the different compilation phases are reviewed as already introduced in section "Compilation Phases and Dependencies", but now from an ASIP per-

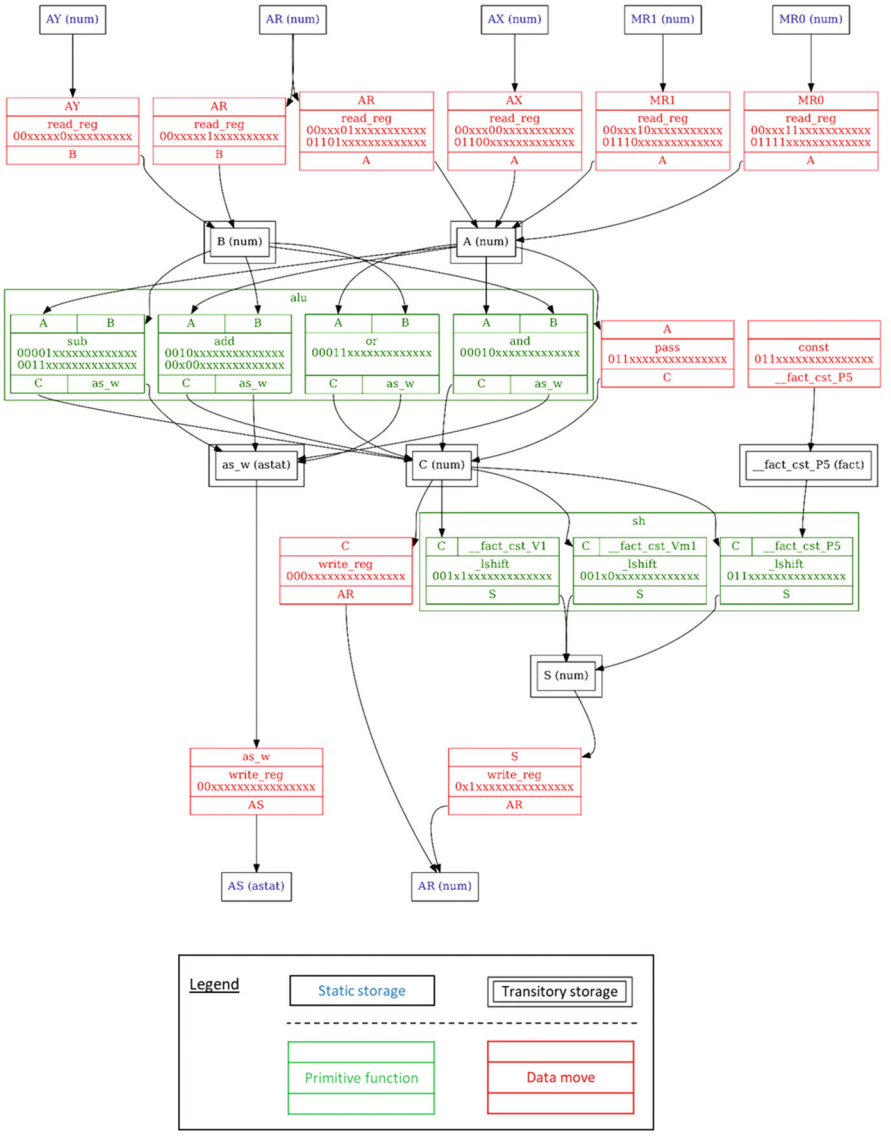


Fig. 9 Portion of an ISG example

spective. Additional requirements are described for these phases, and how they can be handled in a retargetable compiler that targets the broad scope of ASIP architectures described in section "Architectural Scope of ASIPs". Compared to compilers for general-purpose CPUs, the most important differences can be found in the retargetable back end of the compiler. Optimization algorithms in the back end operate on a detailed model of the processor architecture coded in a

processor IR that is automatically constructed from an ADL, such that the compiler retargets instantaneously whenever changes are made to the ADL. The feature of retargetability should however not compromise the generated machine code quality, i.e., the code quality is expected to approximate that of custom-made compilers for the same processor target.

Front End and Middle End

Datatype and Function Support

An important requirement for the compiler front end is that it must enable the use of application-specific datatypes, and of operators and intrinsic functions that operate on such datatypes, in the application source code. These datatypes, operators and functions may either correspond directly to primitive datatypes and primitive functions of the ASIP architecture or expand into primitive types and functions in a subsequent lowering phase in the middle end.

The use of C++ as a source code language is particularly beneficial to cope with these requirements, since application-specific datatypes can be modeled as C++ classes, and the concept of operator overloading that is native to C++ can elegantly be applied to make built-in C++ operators work on these application-specific datatypes. Even without using more advanced object-oriented features of C++, classes and overloading are useful and powerful features to program applications that target ASIPs.

As an example, Fig. 10 shows a fragment of the source code of the DFT prime-factor algorithm to be mapped on the example ASIP of section "Example", using ASIP Designer's retargetable compiler. The example illustrates:

- The use of application-specific datatypes that map on primitive datatypes of the ASIP, such as **vcmplx_t** (6-element vectors of 32-bit complex numbers, each composed of a 16-bit real and a 16-bit imaginary part, totaling 192 bits). Such types are C++ classes defined in a header file.
- The use of overloaded operators, such as the multiply operator in the statements assigning **t0** up to **t4**, which multiplies two **vcmplx_t** types and produces a **vcmplx_t** result, and maps onto the vector multiplication primitive function **vmul** in the ASIP's **VU1** functional unit.
- The use of intrinsic functions, such as the **virdx()** function for a special radix computation, mapping on the **VU2** functional unit.

The above example illustrates that ASIPs with SIMD support are often programmed by explicitly specifying vector datatypes in the application source code, together with overloaded operators and intrinsic functions on those vector datatypes. This is an elegant programming style that is well accepted, because at the time when a SIMD ASIP is conceived, designers often have good insight into how the reference code of their applications can be vectorized. As mentioned in section "Middle End", the compiler research community recently revisited the topic of auto-vectorization. As many ASIPs have SIMD capabilities, auto-vectorization could be a useful feature in a compiler for ASIPs. Current state-of-the-art techniques can

```
void vmod5_s1(vcmplx_t vIn[], vcmplx_t vOut[], int [[chess::storage(m0)]] n)
{
  vcmplx_t c = vTC5;
  vcmplx_t* restrict pIn   = vIn;
  vcmplx_t* restrict pOut  = vOut;
  vcmplx_t [[chess::storage(CM)]]* restrict pTwid = vTwid_1to2;
  idx_t stepn=n;

  for (int k1=0; k1<n; k1++) [[chess::loop_range(1,)]] {
    vcmplx_t i0,i1,i2,i3,i4;
    vcmplx_t t0,t1,t2,t3,t4;
    vcmplx_t o0,o1,o2,o3,o4;
    i0 = pIn[0*stepn];
    i1 = pIn[1*stepn];
    i2 = pIn[2*stepn];
    i3 = pIn[3*stepn];
    i4 = pIn[4*stepn];
    pIn++;

    vmodule5io(i0,i1,i2,i3,i4,t0,t1,t2,t3,t4,select_tc(c,0),cselect_tc(c,2),select_tc(c,0));

    t0 = t0 * pTwid[0*stepn]; o0 = virdx(t0,mrRS);
    t1 = t1 * pTwid[1*stepn]; o1 = virdx(t1,mrRS);
    t2 = t2 * pTwid[2*stepn]; o2 = virdx(t2,mrRS);
    t3 = t3 * pTwid[3*stepn]; o3 = virdx(t3,mrRS);
    t4 = t4 * pTwid[4*stepn]; o4 = virdx(t4,mrRS);
    pTwid++;
    vcmplx_t* restrict p0  = pOut; p0[0*stepn] = o0;
    vcmplx_t* restrict p1  = pOut; p1[1*stepn] = o1;
    vcmplx_t* restrict p2  = pOut; p2[2*stepn] = o2;
    vcmplx_t* restrict p3  = pOut; p3[3*stepn] = o3;
    vcmplx_t* restrict p4  = pOut; p4[4*stepn] = o4;
    pOut++;
  }
}
```

Fig. 10 Application source code fragment of the DFT prime-factor algorithm

handle basic use cases, with code consisting of nested loops with data-independent bounds. For practical use, more research is needed on this topic.

Besides for supporting application-specific datatypes, operators, and intrinsic functions, C++ has also become popular as an application source code language for ASIPs because a growing number of C++ software libraries are becoming available for application domains for which ASIPs are a popular implementation target. Examples include *OpenCV* (computer vision), *TensorFlow Lite for Microcontrollers* (machine learning), and *Eigen* and *Blaze* (vector arithmetic and linear algebra).

Front ends of established compiler frameworks like GCC and especially LLVM are mostly independent of the processor target, and can therefore be used in retargetable compilers for ASIPs. Their existing support of C++ is beneficial in an ASIP context, as explained above. Moreover, leveraging LLVM's modular architecture, the LLVM engineering community recently saw increased interest in research and development of front ends for new domain-specific programming languages. Examples of such languages include *OpenMP* (Chandra et al. 2001) and *OpenCL* (Kaeli et al. 2012) (parallel programming), *Sycl* (Reinders et al. 2021) (heterogeneous system programming), *Rust* (Klabnik and Nichols 2018) (memory-safe functional programming), and *Halide* (Ragan-Kelley et al. 2013) (image processing).

Note though that standard releases of the GCC and LLVM front ends today may not be applicable to ASIPs without modifications. Since GCC and LLVM were originally targeting general-purpose CPUs, several of the typical characteristics of CPUs are reflected in the implementation of these tools, including their front end. See the introduction of section "Architectural Scope of ASIPs" for a list of such characteristics. LLVM's Clang front end has been extended by certain ASIP tool vendors to alleviate the restrictions originating from the general-purpose CPU model (Synopsys 2020).

Source-Code Annotations

The implementation of software code on ASIP architectures often must meet constraints that are imposed by the system in which the ASIP is embedded. One example is *real-time constraints*, where the rate at which certain input data are consumed or output data are produced may not be lower than a specified bound, or the input-output delay (also called latency) may not exceed a specified bound. Another example is *I/O constraints*, i.e., input or output data may have to be stored in designated memory locations or communicated via designated I/O ports.

Retargetable compilers for ASIPs typically support annotations in the application source-code code, by means of which the software programmer can influence the compiler to generate code that meets such constraints.

One type of annotation is the use of standardized keywords in the source code. The code fragment of Fig. 10 shows the use of the *restrict keyword* for pointer variables **pIn**, **pOut,** and **pTwid**. The restrict keyword has become a standardized construct in the C language. By adding this keyword, the programmer hints to the compiler that during the pointer's live range the object that is pointed to will only be accessed by the pointer itself or by a pointer value derived from it. This fact can then be exploited in the data-flow analysis phase in the middle end of the compiler and will eventually result in more freedom in the instruction scheduling phase in the back end. The use of restrict pointers is a common practice in compilers for digital signal processors (DSPs). At the time of writing, LLVM only supported restrict pointers for function arguments. Efforts are under way to generalize this functionality (Dobbelaere 2019), which is important in the context of retargetable compilers for ASIPs.

Compilers may also support nonstandardized user annotations in the source code, also referred to as *pragmas*. To ensure portability of C++ source code between different compilers, it is good practice if compilers allow to describe such annotations in *C++11 attribute* syntax (i.e., between double-square brackets). Compilers that support pragmas in C++11 attribute syntax will simply ignore those pragmas that they do not recognize. The code fragment of Fig. 10 illustrates some of the source-code annotations supported by ASIP Designer's retargetable compiler. The **[[chess::storage()]]** annotation can be used to allocate static variables in the code to a specific memory location or I/O port. With **[[chess::loop_range()]]**, the software programmer can provide hints about the actual range of a *for*-loop counter that cannot be deduced from the source code alone. The compiler can use this

information in the instruction scheduling phase in the back end, to apply more optimal software pipelining (see section "Instruction Scheduling").

Code Selection

In the code selection phase in the back-end stage, the compiler recognizes patterns of operations in the application IR that each can be implemented in a single instruction on the processor target. Such patterns are *operation bundles*. Typically, these bundles take the form of directed acyclic graphs (DAGs), with nodes representing operations and edges indicating data or control dependencies. If the application IR is a CDFG then the bundles are subgraphs within the CDFG. Compiler frameworks store the set of patterns that are supported by the target in a data structure (see LLVM's **TargetLowering** class in section "Processor Intermediate Representation"). In a retargetable compiler that uses a graph-based processor IR, the valid patterns can be determined by analyzing this processor IR, which is also a DAG (see, e.g., the ISG representation of Fig. 9).

Code selection consists of two subtasks. The first subtask, called *matching*, determines possible matches of DAG patterns from the application IR to the processor IR. The second subtask, called *covering*, then selects a set of matched DAGs from the processor IR that covers the complete application IR.

Once code selection has been completed, the application IR is updated such that the selected bundles become single objects. For example, if the application IR uses a DFG representation, the nodes in the DFG now represent bundles, and the edges represent data dependencies between inputs and outputs of these bundles.

DAG matching and DAG covering are both NP-complete problems. To ensure low computation time, compilers apply heuristic methods.

Many traditional compilers use heuristics that transform *DAGs* into *trees*. The advantage is that tree matching and tree covering can be solved in linear time. Both subtasks can now be performed by tree automata for which dynamic programming techniques are frequently used, certainly in case of general-purpose CPU architectures (Aho et al. 1989; Fraser et al. 1992). However, as discussed in section "Specialization", ASIP architectures typically have a heterogeneous storage architecture, and instructions can be subject to register-file sub-range and coupling constraints. For these reasons, code selectors for ASIPs better perform the matching and covering steps directly on DAG patterns.

As an illustration, Fig. 11 shows a few practical examples of DAG patterns that can be implemented on specific instructions in ASIPs. The pattern on the left maps onto an indirect load instruction from a data memory (**DM**) with a postincrement of the address pointer (**ptr**). The address pointer is a common input of the load and postincrement operations, which makes the pattern a DAG instead of a tree. DAG matching ensures that the load and postincrement operations are always performed in the same instruction cycle, as a result of which the pointer must not be stored for more than one cycle and thus only a single pointer register is required for consecutive loads. The pattern on the right maps on a full-precision multiply-add-saturate instruction, as in the expression $\mathbf{d = sat(a*b + c)}$. The multiply operation has two outputs, representing low- and high-order bits. The add operations have two

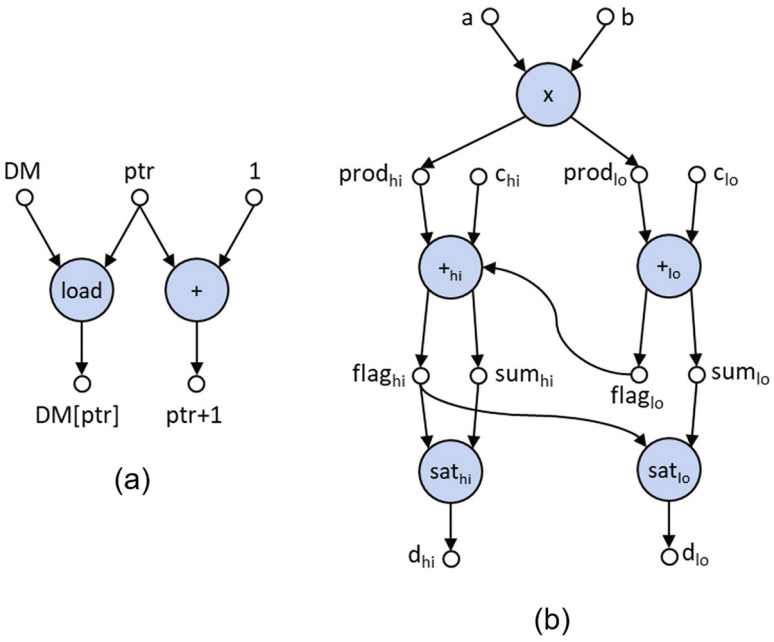


Fig. 11 Example DAG patterns supported by specific instructions: (**a**) Indirect load with address postincrement; (**b**) Multiply-add-saturate instruction on a saturating MAC functional unit with low and high parts

outputs, representing data bits and carryout flags. Such operations with multiple outputs result in DAG patterns.

ASIP Designer uses a code selection technique that directly operates on DAGs, based on the work presented in (Van Praet 1997). Alternative approaches operating on DAGs have been proposed in more recent literature, e.g., (Ertl 1999; Ebner et al. 2008).

Register Allocation

Programming languages like C and C++ distinguish between the following types of variables:

- *Global variables*: These are defined outside of functions, and always exist.
- *Static variables*: These are declared with the *static* keyword, and exist across function calls.
- *Local variables* (also called *automatic variables*): These only exist in the function in which they are defined.

Global and static variables are always stored in memory.

Register allocation is the task of determining a storage location for local variables. The local variables to be allocated are inputs or outputs of operation bundles

that have been created in the code selection phase (section "Code Selection"). They are best kept in registers as much as possible, but due to register capacity limitations the compiler may decide to temporarily move them to memory. This is referred to as *register spilling*. If the processor supports a software stack in memory, spilled variables will be stored in a dedicated part of the software stack frame called the *spill area*.

General-purpose CPUs typically have a single central register file of which all the register fields are equally accessible as inputs (sources) or outputs (destinations) of operation bundles. In that case, what remains to be done is to select a register field in the central register file, for each local variable. This task is referred to as *register assignment*. It is discussed separately in section "Register Assignment".

ASIPs with a heterogeneous storage architecture can have multiple register files of different sizes that are connected to specific inputs or outputs of functional units. Register allocation then becomes a more complex task that must consider aspects like interconnectivity, capacity constraints of each register file, and register-file access restrictions for individual instructions (e.g., sub-range access or register coupling: see the discussion on connectivity and storage architecture in section "Specialization").

Consider the example ASIP of Fig. 12. The three diagrams show alternative register allocations for a local variable that is produced on the shifter unit (**SH**) and consumed as an input to the multiplier unit (**MPY**).

- In the left-hand diagram the variable is stored in the single output register of the shifter, and in a later cycle moved via the result bus to the input port of the multiplier for immediate consumption. While this solution looks straightforward, it may be inefficient in case the result bus is heavily loaded by other instructions. In that case the compiler would have to delay the multiply operation by several cycles, meanwhile blocking the ALU/shifter unit.
- In the middle diagram the variable is again stored in the output register of the shifter. In a subsequent cycle it is moved via an operand bus to an input register of the multiplier. Only in a third cycle it is consumed by the multiplier. Even though this solution requires an additional move instruction, it may be more efficient, if the load of the operand bus is lower than of the result bus.
- The right-hand diagram shows a solution that is beneficial if the production of the variable occurs early, and its consumption occurs late in the program. To avoid

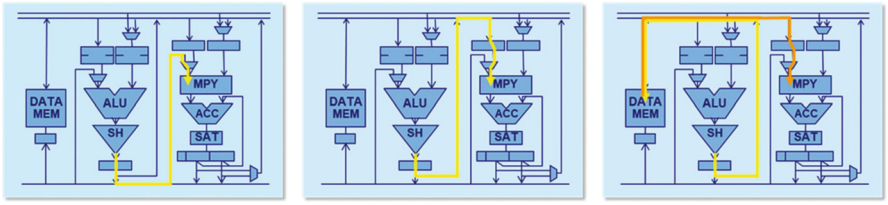


Fig. 12 Three alternative register allocation solutions for the same local variable, in an ASIP with a heterogeneous storage architecture

that it blocks registers during its long live range, the variable is spilled to the data memory, where it can reside during many cycles. Just prior to consumption, it is loaded into an input register of the multiplier and subsequently consumed.

Which alternative is best depends on the application and architecture context.

Register allocation for heterogeneous storage architectures has been addressed in compiler literature. Some approaches extend standard techniques for *register assignment*, which are typically based on graph coloring of an interference graph (see section "Register Assignment"). The idea is that all registers of the heterogeneous architecture continue to be viewed as a central register file, with additional constraints on their use. Heuristic methods are then added to the standard graph coloring techniques to take these constraints into account. Some approaches have been presented that do not use interference graphs. For example, the work in (Scholz and Eckstein 2002) builds a mathematical model that captures the architectural constraints, which is then mapped into a quadratic optimization problem solved with heuristics. Such methods may be applicable to architectures with limited heterogeneity, e.g., DSPs with a central register file for data and a second register file with address registers.

Strongly heterogeneous architectures like ASIPs call for different register allocation techniques. ASIP Designer uses a graph-based *data routing* algorithm to solve the register allocation problem, based on early research presented in (Lanneer et al. 1994). Data routing looks up all possible connection paths in the architecture to route a variable from the point where it is produced (the output of a functional unit) to the points where it is consumed (inputs of functional units). Using certain cost metrics, a specific path is selected. This allocates the variable to each of the storage locations on the selected path. The algorithm uses path search techniques in the ISG model of the processor, combined with branch-and-bound techniques to select efficient solutions.

Scenarios like spilling of variables naturally fit in data routing, as they are simply alternative paths from production to consumption.

Register Assignment

Register assignment refers to the selection of individual registers fields for local variables allocated to register files. This is a well-understood problem in compiler theory. It is assumed that (at least approximative) scheduling of instructions has been done (see section "Instruction Scheduling") so that live ranges of local variables are known.

Variables with overlapping live ranges cannot be assigned to the same register field. Such constraints can be represented as edges in an undirected graph, called *interference graph*, in which the nodes represent the variables. The register assignment problem can then be solved as a graph coloring problem (Chaitin 1982; Briggs et al. 1994).

Register capacity limitations imply that only a limited number of colors may be used. When no solution can be found with the given number of colors, heuristic methods have been described to apply local transformations in the application IR, resulting in reduced interference. Examples include the introduction of spilling, the

recomputation of certain variables, and *live range splitting (*Keith et al. 1998*)*. The latter refers to making copies of variables that each have a shorter live range and therefore can be assigned more effectively.

The heterogeneous storage architecture of typical ASIP architectures further complicates the register assignment task, e.g., to take into account access restrictions within register files as discussed under connectivity and storage architecture in section "Specialization".

Instruction Scheduling

During instruction scheduling, the compiler decides on which control step the operation bundles will be executed. The optimization objective typically is to obtain the lowest cycle count for executing the application, but in certain contexts the smallest code size in program memory may be desired instead.

To deal with phase coupling, some compilers will insert multiple scheduling phases, e.g., computing an approximative schedule prior to register assignment and an exact schedule after register assignment.

If the processor supports instruction-word parallelism, which is the case with most ASIPs, the scheduler is expected to fill the parallel slots with operation bundles as much as possible. The scheduler must respect all data dependencies (i.e., variables can only be consumed after they have been produced) as well as anti-dependencies (i.e., variables cannot be overwritten before their last consumption).

As an illustration, Fig. 13 lists a fragment of the machine code for the DFT prime-factor algorithm, generated by ASIP Designer's retargetable compiler, for

```
VU0                                                
```

the example ASIP introduced in section "Example". For reference, the five slots of the ASIP's parallel instructions are printed at the top. The three-digit numbers at the far left are program counter (PC) values. In addition to radix calculation operation bundles, the **VU2** slot also encodes control operations. At PC value 407 it contains a zero-overhead loop operation (shown in red) with two delay slots and PC end address 429. This means that the loop body ranges from PC values 410 till 429 (shown in red). It can be noted that the compiler was able to fill the parallel slots well. The **DM** slot is entirely filled in the loop body, i.e., there are no "no-operation" (nop) codes, which implies that the loop body has been scheduled optimally.

Aggressive Scheduling of Noninterruptible Code Sequences

Compilers often do not expose the exact instruction pipeline to the scheduler. This may result in a too high cycle count, as is illustrated by the example of Fig. 14. An ASIP with a five-stage instruction pipeline (instruction fetch, instruction decode, register read, execute, and register write) and with instruction-word parallelism over two slots is assumed. Two instructions are shown that can use either slot, respectively, reading from and writing to the same register. Traditionally, an anti-dependency constraint is assumed saying that the instruction writing the new value cannot be scheduled earlier than the instruction reading the old value (depicted by the red edge in the left-hand diagram). If, however, the instruction pipeline is fully exposed, then the compiler can be aware that write operations occur two pipeline stages later than read operations, and therefore the write instruction may already be scheduled two cycles before the read instruction (see right-hand diagram). This scheduling mode is referred to as *aggressive scheduling* in (Goossens 2021).

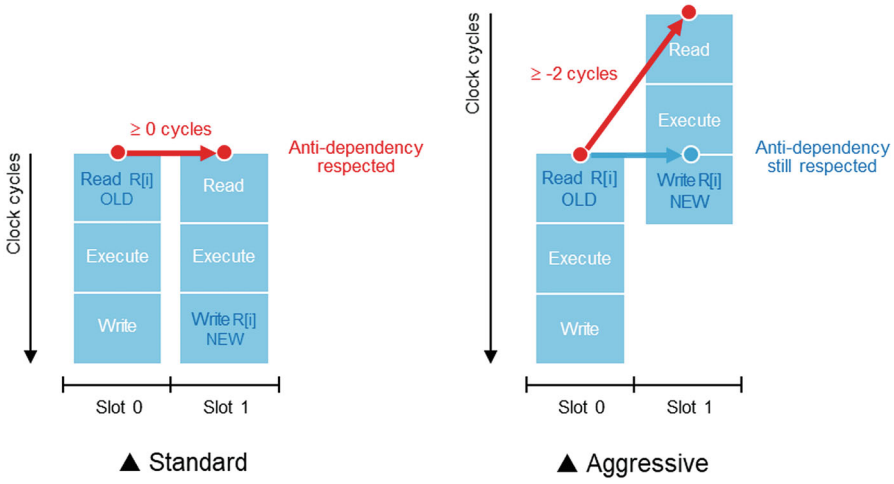


Fig. 14 Standard and aggressive scheduling on an architecture with two-slot instruction-word parallelism

In sections of code that have been scheduled aggressively, the processor cannot accept interrupts. This can be easily understood from the right-hand diagram in Fig. 14: if an interrupt must be serviced between the depicted instruction using slot 1 and the one using slot 0, the anti-dependency can no longer be satisfied. Some compilers automatically ensure that interrupts are masked during the execution of such code sections (Goossens 2021).

Aggressive scheduling reduces the register pressure. As such, for a given small register set, it can produce more compact schedules with a lower cycle count. For this reason, it is frequently used in ASIPs with instruction-word parallelism.

Table 1 compares the cycle count obtained with standard and aggressive scheduling of FFT algorithms of different sizes on the example ASIP of section "Example" (with a four-stage instruction pipeline), using ASIP Designer's compiler (Goossens 2021). The average cycle count gain in these examples is 22%. Higher relative gains can be obtained on ASIPs with deeper pipelines.

Software Pipelining

Software pipelining is a code transformation step in the scheduling phase that is supported by many compilers targeting processors with instruction-word parallelism, when the application code contains *for*-loops (Rau and Glaeser 1981; Lam 1988; Goossens et al. 1989). It applies to both zero-overhead loops and software loops implemented through conditional branching. The compiler will try to move operations from one loop iteration to a subsequent one, where they can fill slots that would otherwise remain unused.

Software pipelining is illustrated in Fig. 15. Fragments of machine code are shown of a covariance function in a wireless communication application, compiled with ASIP Designer's C compiler. The code contains two nested *for*-loops, indicated by the angular arrows. The fragment on the left is for an ASIP with specialized instructions but no instruction-word parallelism. The inner loop requires nine cycles per loop iteration. The fragment on the right is for an ASIP with similar

Table 1 Cycle count comparison for standard and aggressive scheduling of FFT algorithms on example ASIP of section "Example"

FFT Size	Cycle count *standard*	Cycle count *aggressive*	Gain
8	5	5	0%
16	16	14	12%
32	25	20	20%
64	42	29	31%
128	75	52	31%
256	248	180	27%
512	446	328	26%
1024	878	654	25%
2048	1700	1288	24%
4096	4412	3585	19%
Average:			**22%**

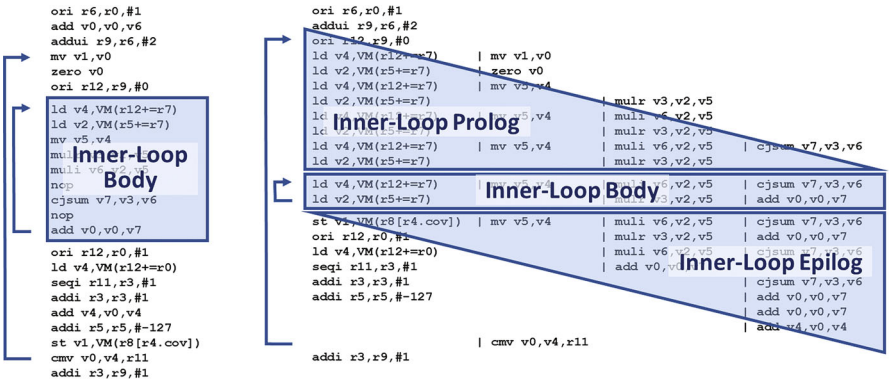


Fig. 15 Compiled machine-code fragments of covariance function with two nested loops, on an ASIP with no instruction-word parallelism (left) and on an ASIP with four slots of instruction-word parallelism where the inner loop has been software pipelined (right)

specialization and with four slots of instruction-word parallelism. The compiler has applied software pipelining to the inner *for*-loop, resulting in a compact schedule of only two cycles per loop iteration. The compiler inserted so-called prolog and epilog code before and after the loop, to ensure correct initialization and termination of the software pipeline. The compiler may try to schedule prolog and epilog code in parallel with application code that already preceded or succeeded the loop body.

Software pipelining is often applied to the innermost loop of each loop nest in the application code. Compilers may also attempt to apply software pipelining at higher levels of loop nests (Muthukumar and Doshi 2001). This is often beneficial because, as illustrated by the triangular shapes in Fig. 15 (right), the slot utilization of a loop's prolog and epilog is typically quite complementary, so that moving epilog code to the beginning of the next iteration of the higher-level loop will result in a higher slot utilization in the higher-level loop body and thus a further cycle count reduction. This is typically at the expense of larger code size, because the compiler now must insert extensive prolog and epilog code for the higher-level loop as well. Alternatively, if the processor supports predicated execution, the instructions in the loop can be predicated to obtain the required prolog and epilog functionality in the initial and final iterations.

Scheduling Techniques

Various instruction scheduling algorithms are being applied in compilers. Examples include:

- *List scheduling:* While its origins go back a long time (Fisher 1979), list scheduling continues to be popular in modern compilers. It is a greedy algorithm that processes control steps from low to high (or the other way round). At every control step a list of yet unscheduled operation bundles is composed that are not data dependent on yet-to-be-scheduled bundles. Bundles from the list are

assigned to the current control step based on a priority function, provided they do not conflict with other bundles already assigned to this control step. Bundles can be conflicting for various reasons: they may use the same functional unit(s), write to the same register port(s), or require incompatible opcode bit settings in the instruction word. Various priority functions have been proposed, and compilers may combine them to find more optimal solutions (De Micheli 1994). More recently, machine learning techniques have been proposed to derive optimized priority functions for list scheduling (Malik et al. 2008).

- *Trace scheduling* is an algorithm that was originally developed for VLIW architectures (Ellis 1986). It places more focus on control-flow aspects of the application. It repeatedly choses a frequently executed path through the application's control-flow graph called a *trace*, schedules it as compactly as possible, and then performs extra bookkeeping to glue adjacent traces consistently together.

Special attention must be paid to the handling of *for*-loops in scheduling. Many schedulers treat loop nests hierarchically, starting by scheduling the inner loops, which are subsequently inserted as macrooperations during scheduling of the next-level loops. Different approaches to *software pipelining* have been proposed and implemented:

- One approach is to use *iterative list scheduling* (Lam 1988; Goossens et al. 1989). In each iteration step of the algorithm, a new list schedule is computed for the loop body. At the end of each step, heuristics are applied to select operation bundles from the obtained schedule and move them to a different loop iteration, resulting in a modified loop body that can potentially be scheduled in fewer cycles. The process repeats until no further reduction of the cycle count for the loop body can be found.
- *Iterative modulo scheduling* is another approach to software pipelining (Rau 1994). This algorithm is also iterative, but instead of trying to reduce the cycle count for the loop body in consecutive iterations, iterative modulo scheduling starts from a precomputed lower bound on the cycle count (referred to as the *minimum initiation interval*) and tries to schedule the loop body within that bound. This may not be successful. In successive iterations, the bound is then increased and the core scheduling algorithm is reapplied until a solution is found. The core scheduling algorithm picks operation bundles based on a priority function and assigns them to control steps within the allowed initiation interval. The control step is chosen depending on the already assigned control steps for other bundles. An important difference with list scheduling is that control step assignments can be undone during this process.

  Extensions to modulo scheduling have been proposed. *Swing modulo scheduling* is an extension that tries to minimize the live ranges of variables in the loop, which helps to improve phase coupling with the register assignment phase in the compiler (Llosa et al. 1996).

Conclusions

The electronics industry of the twenty-first century continues to see strong growth of smart connected products implemented in heterogeneous multicore SoCs. This trend is fueled by the development of novel and more application-specific processor architectures (ASIPs), characterized by increasing amounts of parallelism and specialization. Such advanced processors require efficient design tools, including compilers to develop application software.

While software compilation is an engineering discipline with a long history, today's context brings new challenges for compiler developers. Compilers must be able to exploit the features of highly specialized architectures and must become available quickly, preferably in co-development with the processor architecture so that early compilation of application code can provide feedback to drive architectural decisions. Retargetable compilation is an engineering domain that received renewed attention in view of the above trends and requirements.

This chapter discussed concepts, challenges, and solutions in the domain of retargetable compilation. While visible as an academic research topic in the 1990s, continued developments in retargetable compilation meanwhile resulted in several successful commercial deployments in the context of ASIP design tools. As described in this chapter, retargetable compilers for ASIPs reuse several concepts from the more established field of compiler construction. A selection of references to the compiler construction literature has been provided, which is necessarily incomplete due to the long history of this field. Retargetable compilers however also differ from standard compilers in multiple respects. True retargetable compilers read a processor model expressed in an ADL. Next to a somewhat conventional application IR, retargetable compilers also use a processor IR that is automatically constructed from the ADL. The ADL and the processor IR must be able to capture a wide architectural scope of ASIPs, which can differ significantly from general-purpose processor architectures. Optimizations in a retargetable compiler directly operate on both the application IR and the processor IR, and require sophistication in order to deliver production quality code for each ASIP that can be modeled.

Retargetable compilation is expected to receive continued and renewed interest from the research community in the next decade. Initiatives like RISC-V are promoting the diversification and specialization of processor architectures, and thus of ASIPs. Next-generation smart SoC design projects in industry will see an even stronger need for more advanced ASIP architectures. In addition to supporting advanced processor cores, compilers will have to cope with multicore aspects. Another evolution is the emergence of domain-specific programming languages, with first industry successes in domains like machine learning and advanced vision systems. While domain-specific language front ends are emerging, efficient phase coupling with retargetable compiler back ends will be required to ensure that the overall resulting tool chain can produce production quality code.

Retargetable compilers may be considered as today's most successful incarnation of "hardware/software co-design," a concept that was coined already at the end of

the twentieth century. Continued research in the field of retargetable compilation is expected to contribute to important new advances of the electronics industry in the next decade.

Acknowledgments The authors express their appreciation to the reviewers and to their colleague Sven Wuytack for their constructive feedback on this chapter.

References

Aho AV, Sethi R, Ullman JD (1986) Compilers: principles, techniques and tools. Addison-Wesley, Reading

Aho AV, Ganapathi M, Tjiang SWK (1989) Code generation using tree matching and dynamic programming. ACM Trans On Prog Languages and Systems

Aho AV, Lam MS, Sethi R, Ullman JD (2007) Compilers: principles, techniques and tools. Pearson/Addison-Wesley

Allen R, Kennedy D (1987) Automatic transformations of FORTRAN programs to vector form. ACM Trans. on Prog. Languages and Systems

Allen R, Kennedy K (2001) Optimizing compilers for modern architectures. Morgan Kaufmann

Arm, Arm compiler for embedded, https://developer.arm.com/tools-and-software/embedded/arm-compiler

Arm-KEIL, Embedded development tools, https://www.keil.com

Bik AJC, Girkar M, Grey PM, Tian X (2002) Automatic intra-register vectorization for the Intel architecture. Int. J. of Parallel Programming

Briggs P, Cooper KD, Torczon L (1994) Improvements to graph coloring register allocation. ACM Trans Prog Lang and Systems

Brockmeyer E (2010) Design of an ASIP for DFT/FFT. Technical Report, Target Compiler Technologies

Cadence, Tensilica offerings: development toolchain, https://www.cadence.com/en_US/home/tools/ip/tensilica-ip/technologies.html

Cadence, Tensilica controllers and extensible processors, https://www.cadence.com/en_US/home/tools/ip/tensilica-ip/tensilica-xtensa-controllers-and-extensible-processors.html

CEVA (2020) CEVA-ToolBox Software Development Suite, https://www.ceva-dsp.com/wp-content/uploads/2020/11/07_11_20_ToolBox_Product_Note_EN-V2.pdf

Chaitin GJ (1982) Register allocation & spilling via graph coloring. Proc ACM SIGPLAN Symp on Compiler Construction

Chandra J, Menon R, Dagum L, Kohr D, Maydan D, McDonald J (2001) Parallel programming in OpenMP. Morgan Kaufmann Publishers

Codasip, Codasip RISC-V Processors, Product documentation, https://codasip.com/products/codasip-risc-v-processors

Codasip, Codasip Studio, Product documentation, https://codasip.com/products/codasip-studio/

Cytron R, Ferrante J, Rosen BK, Wegman MN, Kenneth Zadeck F (1991) Efficiently computing static single assignment form and the control dependence graph. Trans Programming Languages and Systems

De Micheli G (1994) Synthesis and optimization of digital circuits. McGraw-Hill

Dennis JB (2011) Data flow graphs. In: Padua D (ed) Encyclopedia of parallel computing. Springer

Dobbelaere J (2019) RFC: Full 'restrict' support in LLVM, https://lists.llvm.org/pipermail/llvm-dev/2019-March/131127.html

Ebner D, Brandner F, Scholz B, Krall A, Wiedermann P, Kadlec A (2008) Generalized instruction selection using SSA-graphs. Proc. ACM SIGPLAN/SIGBED Conf. on Lang., Compilers and Tools for Embedded Systems

Ellis JR (1986) Bulldog: a compiler for VLIW architectures. The MIT Press

Ertl MA (1999) Optimal code selection in DAGs. Proc. ACM/SIGPLAN-SIGACT Symp. on Principles of Prog. Lang
Fischer CN, LeBlanc RJ (1991) Crafting a compiler with C. Benjamin/Cummings
Fisher JA (1979) The optimization of horizontal microcode within and beyond basic blocks: an application of processor scheduling with resources. Ph.D Thesis, New York Univ
Fraser CW, Hanson DR (1995) A retargetable C compiler: design and implementation. Pearson Education
Fraser CW, Hanson DR (2001) The Lcc 4.x code generation interface. Technical Report, Microsoft Research
Fraser CW, Henry RR, Proebsting TA (1992) BURG – fast optimal instruction selection and tree parsing. ACM SIGPLAN Notices
Free Software Foundation, GCC, the GNU Compiler Collection, https://gcc.gnu.org
Goossens G, Vandewalle J, De Man H (1989) Loop optimization in register-transfer scheduling for DSP-systems. Proc ACM/IEEE Design Autom Conf
Goossens G (2021) Under the Hood of ASIP designer – application-specific processor design made possible by tool automation. Synopsys DesignWare Technical Bulletin Q4/21
Gough BJ (2004) An introduction to GCC: for the GNU compilers GCC and G++. Network Theory
Griffith A (2018) GCC: the complete reference. McGraw-Hill
Grune D, Jacobs CJH (1990) Parsing techniques: a practical guide. Ellis Horwood
IAR Systems, Embedded development, https://www.iar.com/knowledge/learn/programming
International Workshop on Software and Compilers for Embedded Systems (SCOPES), https://scopesconf.org
Kaeli D, Mistry P, Schaa D, Zhang DP (eds) (2012) Heterogeneous computing with OpenCL 2.0. Morgan Kaufmann Publishers
Keith D, Cooper L, Simpson T (1998) Live range splitting in a graph coloring register allocator. Lecture Notes in Computer Science, volume 1383, Springer Verlag
Klabnik S, Nichols C (2018) The Rust programming language. No Starch Press
Lam M (1988) Software pipelining: an effective scheduling technique for VLIW machines. Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation
Lanneer D, Cornero M, Goossens G, De Man H (1994) Data routing: a paradigm for efficient data-path synthesis and code generation. Proc Int Symposium on High-Level Synthesis
Lattner C, Adve V (2004) LLVM: a compilation framework for lifelong program analysis & transformation. Proc. 2004 Int. Symp. on Code Generation and Optimization (CGO'04), Palo Alto
Lattner C, Amini M, Bondhugula U, Cohen A, Davis A, Pienaar J, Riddle R, Shpeisman T, Vasilache N, Zinenko O (2021) MLIR: Scaling compiler infrastructure for domain specific computation. IEEE/ACM Int. Symp. on Code Generation and Optimization (CGO)
Leupers R, Marwedel P (1998) Retargetable code generation based on structural processor description. Des Autom Embed Syst
Leupers R, Marwedel P (2013) Retargetable compiler technology for embedded systems: tools and applications. Springer
Levine JR (2009) Flex & Bison. O'Reilly
Levine JR, Brown D, Mason T (1992) Lex & Yacc. O'Reilly
Llosa J, González A, Ayguadé E, Valero M (1996) Swing modulo scheduling: a lifetime-sensitive approach. Proc. Conf. on Parallel Architectures and Compilation Techniques
LLVM Project, The LLVM compiler infrastructure, https://llvm.org
LLVM Project, Clang: a C language family frontend for LLVM, https://clang.llvm.org
LLVM Project, Auto-vectorization in LLVM, https://llvm.org/docs/Vectorizers.html
LLVM Project, Writing an LLVM backend, https://llvm.org/docs/WritingAnLLVMBackend.html
Malik AM, Russel T, Chase M, van Beek P (2008) Learning heuristics for basic block instructions scheduling. J of Heuristics, December 2008
Marwedel P, Goossens G (1995) Code generation for embedded processors. Kluwer Academic Publishers

Moons B, De Brabandere B, Van Gool L, Verhelst M (2016) Energy-efficient ConvNets through approximate computing. IEEE Winter Conf. on Applications of Computer Vision
Morgan R (1998) Building an optimizing compiler. Digital Press
Muchnick SS (1997) Advanced compiler design and implementation. Morgan Kaufman, San Francisco
Muthukumar K, Doshi G (2001) Software pipelining of nested loops. Pr Int Conf on Compiler Construction
Nowak L (1987) Graph based retargetable microcode compilation in the MIMOLA design system. Proc 20th Annual Workshop on Microprogramming
NXP Semiconductors, CodeWarrior Embedded Software Development Tools, https://www.nxp.com/design/software/development-software/codewarrior-development-tools:CW_HOME
Parr T (2007) The definitive Antlr reference: building domain-specific languages. Pragmatic Bookshelf
Ragan-Kelley J, Barnes C, Adams A (2013) Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. Proc ACM SIGPLAN Conf on Programming Language Design and Implementation
Rau BR (1994) Iterative modulo scheduling: an algorithm for software pipelining loops. Proc 27th Annual Workshop on Microprogramming
Rau BR, Glaeser CD (1981) Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. Proc 14th Annual Workshop on Microprogramming
Reinders J, Ashbaugh B, Brodman B, Kinsner M, Pennycook J, Tian X (2021) Data parallel C++: mastering DPC++ for programming of heterogeneous systems using C++ and SYCL. Apress
Sanghavi H, Andrews N (2008) TIE: An ADL for designing application-specific instruction-set extensions. In: Mishra P, Dutt N (eds) Processor Description Languages. Morgan-Kauffman
Scholz B, Eckstein E (2002) Register allocation for irregular architectures. Proc. Conf. on Lang, Compiler and Tools for Embedded Systems/Software and Compilers for Embedded Systems
Stallman RM (1987) GNU C compiler beta test release. Newsgroup comp.lang.c.
Synopsys (2020) Technology Feature: LLVM extended – A C/C++ compiler frontend for application-specific processors. ASIP eUpdate
Synopsys (2022) Technology feature: wide scope of RISC-V ASIP models ready for ASIP accelerator development. ASIP eUpdate
Synopsys, DesignWare ARC MetaWare Development Toolkit, https://www.synopsys.com/metaware
Synopsys, ASIP Designer, https://synopsys.com/dw/ipdir.php?ds=asip-designer
Texas Instruments, Optimizing C/C++ compilers for our programmable embedded processors, https://www.ti.com/design-resources/embedded-development/ccs-development-tools/compilers.html
Van Praet J (1997) Processor modelling and code generation techniques for retargetable compilation. PhD thesis, KU Leuven
Van Praet J, Lanneer D, Geurts W, Goossens G (2001) Processor modelling and code selection for retargetable compilation. ACM Tr. Design Autom. of Electronic Systems
Van Praet J, Lanneer D, Geurts W, Goossens G (2008) nML: a structural processor modelling language for retargetable compilation and ASIP design. In: Mishra P, Dutt N (eds) Processor description languages. Morgan-Kauffman
Wang S, Kanwar P (2019) Bfloat16: the secret to high performance on cloud TPUs. Google Cloud Blog
Waterman A, Asanović K (2019) The RISC-V instruction set manual, volume I: unprivileged ISA. RISC-V Foundation
Waterman A, Asanović K, Hauser J (2021) The RISC-V instruction set manual, volume II: privileged architecture. RISC-V Foundation

Part VI
Test and Verification

Verification and Its Role in Design of Modern Computers

33

Sayak Ray

Contents

Introduction 1192
Formal Verification, Simulation, and Emulation 1193
Outline of the Section 1194
Section Organization 1195
Bit-Level Model Checking Algorithms 1195
C-to-RTL Equivalence Checking 1195
Symbolic Simulation 1197
Mechanical Theorem Proving 1197
Versatile Binary-Level Concolic Testing 1198
Information Flow Analysis 1198
Verification of Quantum Circuit Design Flows 1199
Discussion 1199
Conclusion 1201
References 1201

Abstract

Computer-aided verification is an integral and crucial step in the design process of modern computing systems. This section sheds light on different computer-aided verification techniques that have been designed and optimized to solve specific verification challenges that were encountered by designers of modern computers. Computer-aided verification is an active area of research and not all verification topics are discussed in this section; instead, the section highlights the verification topics that are at the center stage in the application world – either because of their use in the day-to-day work of verification engineers engaged in verifying various design fragments or because of their impact of some emerging

S. Ray (✉)
Intel Corporation, San Jose, CA, USA
e-mail: sayak.ray@intel.com

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_33

aspect of computer architecture. The chapters under this section should help a computer architect understand the challenges and opportunities in verifying complex system, and in turn, this could influence his or her design decisions.

Keywords

Verification · Model checking · Equivalence checking · Symbolic simulation · Theorem proving · Information flow analysis

Introduction

In the standards of the computing industry, there is a nuanced difference between the terms *validation* and *verification*. As per IEEE standard 1012, validation is the process that provides evidence whether a product solves the right problems to satisfy user needs; on the other hand, verification is the process that provides evidence that a product meets its requirements and standards (IEEE standard for system 2017). In other words, validation ensures that a product solves the right problem and verification ensures that the product solves the problem in a right way. Despite this subtle difference, these two terms are often used interchangeably in practice to mean the set of activities performed to verify that a product satisfies its specifications and meets the needs of the user. In this chapter, the term *verification* will be used to represent such a set of activities. While the definition of verification does not stipulate which technique should be used to perform the task, the term is often associated with static techniques that are performed at the design compilation time. In contrast, techniques that are dynamic in nature, i.e., that involve running the design against candidate inputs and checking whether the outputs meet expectations, are associated with the term *testing*.

Verification is a crucial step in the design cycle of any computing product. The world has witnessed catastrophic failures of computers caused by escapes in verification, resulting in loss of resources such as time, money, and even human lives. The growing complexity of computers indicates that failures of comparable magnitude can happen again unless computer designers adopt the highest standards of verification in their design process. It is perhaps beyond any dispute that only automation can handle the level of complexity involved in verification of modern computers. Automated verification is a computationally hard problem though. In general, exhaustive verification amounts to exhaustive search over a design state space that is inherently exponential in size. Covering such a state space using a brute-force search algorithm hardly works for a real-world design of even moderate size. This challenge has inspired researchers to develop specialized algorithms, heuristics, and abstraction techniques and their optimized implementations to push the boundary of automated verification. As a result, the field has grown both within and beyond the academic research community. Along with academia, a vibrant sector has emerged in the Electronic Design Automation (EDA) industry to provide verification software packages to the semiconductor design houses. The design

houses have integrated such third-party software along with their home-grown automated verification methodologies in various phases of their design cycles.

Despite such progress, automated verification is far from being a "solved problem." The primary reason behind this is the complexity of modern computers. To meet this challenge, verification algorithms have to be more efficient and scalable. But it can be argued that clever design of verification algorithms alone cannot bridge the gap—a better understanding of the design space is needed so that it can be effectively partitioned into smaller modules, which can be handled by the automation tools. For this, a deeper synergy between the design community and the verification community is needed. Both the teams need to understand the other team's language, prerequisites, and constraints. Architects and micro-architects can benefit from rigorous verification by adopting it not only for demonstrating their designs' correctness but also as a methodology for designing modules with *verifiable interfaces*. This improves composability of modules into System-on-Chips (SoC), thereby improving verifiablity of the latter. This seems to be a promising approach for reducing bug escapes while developing complex computers under time-to-market pressure. It is, therefore, useful for the architecture community to understand how the available verification techniques work, what are their limitations, and how they may collaborate with the verification teams to improve the coverage of verification for their designs.

With a vision of bringing the communities of architects and validators closer, this section presents a spectrum of verification techniques that have become the preferred solutions for a practicing verification engineer for different verification problems. Computer-aided verification has a rich theory whose foundation was laid by the pioneers such as Alan Turing, Alonzo Church, and Kurt Gödel in 1930s. The subject got a major boost since 1970s with the seminal contribution by Amir Pnueli, Edmund Clark, Alan Emerson, Josef Sifakis, and their contemporary researchers. An interested reader may want to consult books such as Clarke et al. (2018a,b) and BaierKatoen and Katoen (2008) for comprehensive discussions on the theoretical underpinning of computer aided verification. This section of the handbook would rather focus on some particular aspects of automated verification that have been distilled over time into practical and useful techniques that are being actively deployed by the semiconductor industry in their design cycles.

Formal Verification, Simulation, and Emulation

Computers are composed of hardware and software; the latter includes application software and system software. While bug escape in any of these components can lead to failure of the whole system, computer architects are usually concerned with bugs in hardware and system software. This section would, therefore, focus on techniques that are practiced to verify hardware, system software, and their interactions. As hinted at the beginning of this section, verification techniques are broadly classified into two types: static techniques and dynamic techniques. Static techniques are performed during the compilation of the design source code

and does not involve running the code using test inputs. Static techniques mostly apply mathematical reasoning on the structure of the design to infer its logical correctness. Due to their strong association with formal logic of programming, many of these techniques are classified under *formal verification* techniques. Dynamic techniques, on the other hand, involve running the source code of the design against a designated set of inputs and the resulting outputs are checked against the expected behavior. Such techniques are broadly called *testing*. Computers are subjected to two types of testing—*simulation testing* and *emulation testing*. Computer hardware is developed in high-level hardware description languages such a Verilog, VHDL, or SystemVerilog. These languages come with their run-time simulation environment so that modules developed in these languages can be simulated using test inputs. This type of dynamic verification of hardware is called simulation testing. This offers a flexible, low-cost, and quick way of verifying whether a module is behaving in the right way. Simulation is a great way of finding bugs in a module. However, if it does not find a bug with the designated set of test vectors, it does not guarantee that a module is bug free in general. Also, RTL-level simulation becomes prohibitively slow when applied at the system level. To alleviate the latter problem, practitioners resort to emulation testing where the RTL-level design is synthesized onto reconfiguration hardware platforms, and the resulting hardware configuration is executed against the test inputs. Emulation can achieve significant speedup so that it can scale for large system-level designs, which are beyond the scope of simulation.

Outline of the Section

This section will discuss a collection of carefully selected verification techniques that emerged from academic research community and embraced by the semiconductor industry due to their practical impact. Such techniques are bit-level model checking, C-to-RTL equivalence checking, symbolic simulation, mechanical theorem proving, and concolic testing. Some of them are formal verification techniques, others are simulation oriented. Two other techniques will be discussed, which are relatively new area of research. They are information flow analysis and verification of quantum computers. These two chapters are included for two separate reasons. Security has emerged as a key design constraint in the recent times due to dramatic increase in the cyberattacks on the computing systems. Verification of security claims of a system requires information flow analysis that may not be covered by the techniques discussed so far. In the chapter on information flow analysis, some special techniques will be discussed, which are essential for security verification. Quantum computing is an alternative computer architecture that promises massive computing power that is suitable to solve many problems that are beyond the capacity of today's computer architecture. Researchers have made immense progress in designing quantum algorithms and architecture. However, due to their inherent difference with traditional architecture, traditional verification techniques are not suitable for such architecture. Our final chapter is thus devoted to the discussion of a set of radically different techniques that are suitable for

verifying such emerging architecture. In the remainder of this chapter, high-level introductions to subsequent chapters are provided. Hopefully, this will offer the reader a glimpse of what is to follow.

Section Organization

Bit-Level Model Checking Algorithms

In hardware verification, *model checking* is a process that decides whether a temporal logic formula is satisfied by a finite state machine. The temporal logic formula represents a property that is to be verified on the finite state machine. The actual algorithmic description of the process depends on the formalisms used to present the property and the finite state machine. The general theory of hardware model checking was laid out based on the Kripke Structure formalism for the finite state machine and for temporal logics such as LTL, CTL, and CTL* (Clarke et al. 2018a). Since this theory works on hardware circuits at their bit-level representations, it is called *bit-level model checking*. The first practical breakthrough for hardware model checking came in the form of *symbolic model checking* (SMC) that helped model checking scale on real-world control and data path circuits (Burch et al. 1992). The key behind the success of SMC was in representing the state space of a circuit symbolically using a data structure called *binary decision diagram* or BDD (Bryant 1986). The second breakthrough in efficiency of model checkers came through advancements in Boolean satisfiability solvers *a.k.a.* SAT solvers (Marques-Silva and Sakallah 1999; Moskewicz et al. 2001; Eén and Sörensson 2003). This was a paradigm shift from SMC as this required the circuit to be represented as a Boolean formula and use of a SAT solver to discharge the verification objective. Ingenious use of SAT solvers through innovative formulations of the model checking problem has pushed the boundary of scalability even further (Biere et al. 1999; McMillan 2003; Bradley 2011). Due to high degree of automation and the ability of generating a counterexample when a property is not satisfied by the hardware, bit-level model checking has become the workhorse for formal verification in the hardware industry. ▸ Chapter 34, "Bit-Level Model Checking" outlines the innovations that has elevated model checking from a theoretical discourse to a practical automation solution for formal verification of hardware circuits. The chapter reviews the major milestones in the history of safety and liveness verification algorithms, with a special focus on the scalable solutions that are relevant for the semiconductor industry. Its exposition demonstrates the power of *inductive reasoning* for proving correctness of hardware systems.

C-to-RTL Equivalence Checking

As the name suggests, *equivalence checking* is an automated process of establishing logical equivalence between two circuits. Let us consider two n-input, 1-output

combinational Boolean circuits C_1 and C_2. If C_1 represents Boolean function $f : \mathbb{B}^n \rightarrow \mathbb{B}$ and C_2 represents Boolean function $g : \mathbb{B}^n \rightarrow \mathbb{B}$ where $\mathbb{B} = \{0, 1\}$ is the Boolean domain, C_1 and C_2 are called equivalent if and only if for any bit-vector $\langle b_1, \ldots, b_n \rangle \in \mathbb{B}^n$, $f(b_1, \ldots, b_n) = g(b_1, \ldots, b_n)$. C_1 and C_2 may be conjoined by combining their outputs through an XOR gate and merging their corresponding inputs to derive a new function $h : \mathbb{B}^n \rightarrow \mathbb{B}$ as shown in Fig. 1. This construction of h is called *miter construction*. It can be argued that C_1 and C_2 are equivalent if h is always 0. A SAT solver can be used to prove whether h is unsatisfiable, hence proving if C_1 and C_2 are equivalent. This notion of equivalence can be extended to sequential Boolean circuits as well.

In the above definition, both C_1 and C_2 are presented as Boolean circuits. While this is a fundamental problem that is routinely solved using SAT solvers in logic synthesis and verification engines in practice, the notion of equivalence, however, goes beyond the abstraction boundary. For example, C_1 could be a software model of a digital signal processing algorithm, and C_2 could be its hardware implementation that will be actually fabricated on silicon. Functional equivalence can still be defined between their input–output relations. In fact, such equivalence is routinely tested in practice when a new architectural concept/algorithm is first encoded in a high-level language such as C, C++, or SystemC and later transformed in some hardware description language. Since it is easier to verify high-level models, designers usually have more confidence on the C/C++ models. Functional equivalence checking between the C/C++ model and its hardware implementation

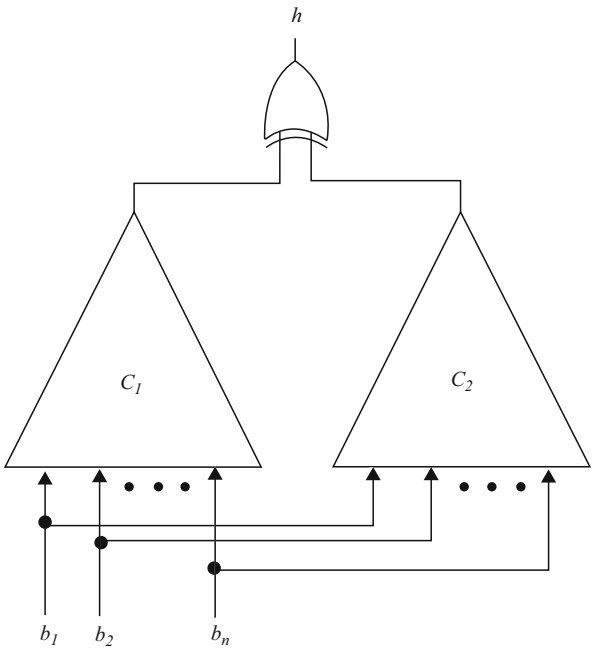


Fig. 1 Miter construction for equivalence checking

helps gain similar confidence on the latter. Due to the popular choice of high-level modeling language, this problem is commonly called C-to-RTL equivalence checking in the industry. Chapter "C-to-RTL Equivalence Checking" discusses various challenges and opportunities of this important problem and available solutions to it.

Symbolic Simulation

Model checking and equivalence checking are two major pillars of automated verification of hardware systems. They are general purpose and fully automated, require minimal human intervention, and produce counterexamples that expose the root cause of violations. This has led to their successful adoption in industry—multiple commercial vendors provide state-of-the-art model checkers and equivalence checkers today and design houses have absorbed them successfully in their verification flows. While widely successful in their practical applications, their performance is somewhat limited by the generality of their back-end algorithms. For example, commercial model checkers support verification of any temporal property written in System Verilog Assertion (SVA) on any Boolean circuit written in Verilog, SystemVerilog, or VHDL. While this generality is one of the strengths of these model checkers, it may result in poor scalability. Specialization often comes to rescue in such situations. Specialization can be achieved in terms of both verification algorithm and property specification. Symbolic simulation, which is the topic of ▸ Chap. 36, "Verification of Arithmetic and Datapath Circuits with Symbolic Simulation", is one such specialized technique applied to the problem of verifying arithmetic and data-path circuits. While general-purpose model checkers can handle a much larger class of properties, symbolic simulation restricts itself to the verification of invariants within finite time window offset. In return, symbolic simulation scales on integer and floating-point execution pipelines of real-world processor designs, which are beyond the reach of general-purpose model checkers.

Mechanical Theorem Proving

The biggest advantage of the techniques discussed so far is that they are automatic. The user is supposed to provide the description of the system (hardware circuit) and the properties, and the algorithm automatically detects whether the properties are satisfied by the system. If not, the algorithm also produces a counterexample showing the root cause of falsification. While highly automated, these techniques suffer from scalability issues. Mechanical theorem proving sits on the other side of the spectrum providing less automation but more control on the convergence of the proof process. In this interactive process, the verification obligation is discharged as a mathematical formula and proved using standard mathematical technique such as induction, term rewriting, simplification, and generalization. The proof searching is done by a computer program called *theorem prover*. A theorem prover attempts

to derive a proof of the verification obligation (the theorem) using the rules of the underlying logic. However, it may not be able to derive a proof for a given theorem, and the human expert needs to intervene and supply some intermediate lemma(s), which might be easier target(s) for the theorem prover and proof of which might help the theorem prover to derive the proof of the original theorem. By manually structuring and decomposing the verification problem, the user can guide the theorem prover into proofs of very complex systems. ▸ Chapter 37, "Microprocessor Assurance and the Role of Theorem Proving" demonstrates how theorem prover can be used to verify both architectural and microarchitectural properties of a microprocessor. As a case study, it demonstrates theorem proving based verification of x86 architecture.

Versatile Binary-Level Concolic Testing

The topics discussed so far focus on various aspects of formal verification of hardware systems. While successful on hardware, these techniques are not so effective on system software. These are low-level software that interact closely with the underlying hardware and play a crucial role in defining the overall function of the system. Their control flows are often simple, but they handle numerous complex data structures and convoluted logic of operation. Effective verification techniques for such low level software are an active area of research that tries to meet the challenge of scalability and coverage. Due to their large state space, full proof of correctness using techniques such as model checking does not seem feasible for such software. On the other hand, random testing does not offer any coverage guarantee. In the face of these challenges, researchers have demonstrated that concolic testing offers an effective solution to this problem. ▸ Chapter 38, "Versatile Binary-Level Concolic Testing" shows how concolic testing has been used to verify correctness of system software such as CTOS Linux kernel modules and hardware/software co-validation of systems-on-chips.

Information Flow Analysis

Information flow analysis refers to the process of analyzing how information flows from one point to another point in a computing system. It is also known as Information Flow Tracking (IFT). Many system security questions can be formulated as IFT problems. While this concept was known since Denning (1976), unprecedented proliferation of cybersecurity attacks in recent years has elevated its significance as a defense mechanism. With the advent of micro-architectural side channel attacks such as Spctre (Kocher et al. 2019) and Meltdown (Lipp et al. 2018), IFT is gaining attention from the architecture and automation research communities. While treatment to IFT was more theoretical and software security centric in the past (Cohen 1977, 1978; McLean 1992; Volpano et al. 1996; Sabelfeld and Myers 2003), novel and scalable IFT tools are now being developed both in academia and

industry that can be deployed for hardware security analysis at the register transfer level (RTL) (Hu et al. 2021).

IFT, however, is a fundamentally different problem when compared to traditional functional verification. While the latter maps to verification of *trace properties*, the former belongs to verification of *hyperproperties* (Clarkson and Schneider 2010). Therefore, traditional static analysis technique such a model checking or dynamic analysis technique such as simulation will not directly solve the IFT problem. Generalized algorithms for verifying hyperproperties have been proposed (Finkbeiner et al. 2015). However, it is shown that IFT problem can be solved with less expensive algorithms of self-composition and equivalence checking by mapping IFT to 2-safety problem (Terauchi and Aiken 2005). On the dynamic side, researchers have shown that simulation based verification can be leveraged for IFT problems by augmenting the RTL with flow tracker logic (Hu et al. 2021). Interestingly, the same completeness and scalability trade-offs exist among the static and dynamic techniques for IFT as well. ▶ Chapter 39, "Information Flow Verification" discusses these techniques, their trade-offs, and the current research trends in IFT in detail.

Verification of Quantum Circuit Design Flows

The chapters so far collectively describe the techniques that are being used in the semiconductor industry today for its verification needs. The verification techniques thus described are conceived and optimized for mainstream synchronous hardware design flow. While it is not an exaggeration to say that almost entire semiconductor industry revolves around synchronous hardware design today, quantum computing is emerging as a promising alternative to classical semiconductor technology. In tandem with the development of quantum architecture and algorithms, it is imperative to develop verification techniques for quantum hardware as well. ▶ Chapter 40, "Verification of Quantum Circuits" describes various algorithms that are developed to verify quantum hardware.

Discussion

Computer-aided verification is a vast research area. In this section, a few selected topics have been covered that are being actively practiced in the semiconductor industry to verify computer hardware designs at various levels of abstraction. Such verification methodologies have been relentlessly developed and perfected by numerous researchers and engineers over the past four decades, and the effort is still continuing. The topics are shortlisted for this section mainly based on their technical maturity, generality, practical use, and wide-spread impact. Topics such as *Bit-level Model Checking*, *C-to-RTL Equivalence Checking*, *Theorem Proving*, and *Verification of Datapath Circuits* are great examples of technically mature fields. On the other hand, topics such as *Information Flow Verification* and *Verification of*

Quantum Circuits are relatively new research paradigms; they are included in this section for their immense practical importance. We are poised to see more progress in those fields in the near future, and we intend to capture them in our subsequent editions.

▶ Chapter 38, "Versatile Binary-Level Concolic Testing" is the only chapter in this section that is related to dynamic or run-time verification techniques. The other chapters deal with static or compile-time verification techniques. Such static techniques offer exhaustive and principled verification solutions, which are essential for designing complex systems. It can be argued that architects need to embrace more formal methods for tackling design complexities as well as workforce complexities that are unavoidable for big projects. However, poor scalability and significant learning curve are two major impediments against a wider adoption of formal methods, and this is where dynamic techniques such as simulation and emulation come to rescue. Simulation can test designs whose size is beyond the reach of formal methods and emulation can achieve that at a hardware speed. Any topic on hardware-level simulation or emulation is not included in this section though. It is primarily because there are many authoritative references already available on these topics, which have covered almost all established aspects (Wang et al. 2006). Nevertheless, certain dynamic techniques have seen recent surge in interest. Hardware fuzzing is one such area where constraint random simulation techniques are being revisited for hardware security assurance (Laeufer et al. 2018). We are hoping to cover new results in such contemporary research topics in subsequent edition of the book.

Computer-aided verification is far from being a solved problem. As designs are becoming more complex and time-to-market is shrinking, the gap between what automated verification can deliver and what the semiconductor industry needs is certainly not closing. This is why researchers are actively looking for advanced techniques and methodologies for accelerated sign-off, shift-left for verification deployment, and more verification coverage. While many of these advanced techniques are still in the realm of academic research, it is worthwhile to call out some of them as they have the potential to impact the industry in a positive way. One such area is machine-readable specification with instruction set modeling. For boosting confidence in the quality of industrial designs, the architecture community is embracing a change in how they specify the instruction set architecture (ISA). Traditionally, ISAs have been subjected to limited automated analysis, if any. However, the practice is changing as researchers are showing the benefits of capturing ISA in a machine-readable format that can be passed through a series of formal analyses such as type checking or model checking (Reid 2016; Huang et al. 2018; Armstrong et al. 2019). Beyond machine-readable ISA specification, architectural modeling has gained significant momentum in the past decade for verification of memory consistency (Trippel et al. 2017; Zhang et al. 2018) as well as micro-architectural side channels (Trippel et al. 2019; Hossain et al. 2020). While architectural models have been manually crafted by the researchers, recent research is showing that they can be generated automatically as well (Hsiao et al. 2021). Overall, it is a much-needed trend in research that would bring more automation in reasoning about computer architecture.

Another system-level verification paradigm that is gaining traction in the research community is hardware/software co-simulation. While pure hardware simulation and emulation have become routine operations today, hardware/software co-simulation is far from having a standardized solution. The importance of this topic, however, is growing considerably as unchecked hardware/software interactions can lead to insidious functional and security bugs in SoCs. We are anticipating that new research results will consolidate the current industry practices, which are rather ad-hoc in nature. For a recent survey on relevant techniques, reader is referred to Fasano et al. (2021).

Conclusion

This section is designed to cover a collection of verification techniques that have observed a plethora of practical applications in the semiconductor industry and a well-rounded computer architect should have familiarity with them. This field is certainly evolving, which is evident from a couple of chapters which cover more emerging topics compared to more mature topics covered in the other chapters. Last but not the least, a short sampler of a few promising domains is provided above where research is actively being conducted by the automated verification community. We are looking forward to covering these topics in length in the subsequent edition of this book as more impactful results accumulate.

References

Armstrong A, Bauereiss T, Campbell B, Reid A, Gray KE, Norton-Wright R, Mundkur P, Wassell M, French J, Pulte C et al (2019) ISA semantics for armv8-a, risc-v, and cheri-mips

Baier C, Katoen J-P (2008) Principles of model checking. MIT Press

Biere A, Cimatti A, Clarke E, Zhu Y (1999) Symbolic model checking without BDDs. In: International conference on tools and algorithms for the construction and analysis of systems. Springer, pp 193–207

Bradley AR (2011) Sat-based model checking without unrolling. In: International workshop on verification, model checking, and abstract interpretation. Springer, pp 70–87

Bryant RE (1986) Graph-based algorithms for boolean function manipulation. Comput IEEE Trans 100(8):677–691

Burch JR, Clarke EM, McMillan KL, Dill DL, Hwang L-J (1992) Symbolic model checking: 1020 states and beyond. Inf Comput 98(2):142–170

Clarke EM Jr, Grumberg O, Kroening D, Peled D, Veith H (2018a) Model checking. MIT Press

Clarke EM, Henzinger TA, Veith H, Bloem R et al (2018b) Handbook of model checking, vol 10. Springer

Clarkson MR, Schneider FB (2010) Hyperproperties. J Comput Secur 18(6):1157–1210

Cohen E (1977) Information transmission in computational systems. In: Proceedings of the sixth ACM symposium on operating systems principles, pp 133–139

Cohen ES (1978) Information transmission in sequential programs. Found Secure Comput 297–335

Denning DE (1976) A lattice model of secure information flow. Commun ACM 19(5):236–243

Eén N, Sörensson N (2003) An extensible SAT-solver. In: International conference on theory and applications of satisfiability testing. Springer, pp 502–518

Fasano A, Ballo T, Muench M, Leek T, Bulekov A, Dolan-Gavitt B, Egele M, Francillon A, Lu L, Gregory N et al (2021) SOK: enabling security analyses of embedded systems via rehosting. In: Proceedings of the 2021 ACM Asia conference on computer and communications security, pp 687–701

Finkbeiner B, Rabe MN, Sánchez C (2015) Algorithms for model checking hyperltl and hyperctl *. In: International conference on computer aided verification. Springer, pp 30–48

Hossain N, Trippel C, Martonosi M (2020) Transform: formally specifying transistency models and synthesizing enhanced litmus tests. In: 2020 ACM/IEEE 47th annual international symposium on computer architecture (ISCA). IEEE, pp 874–887

Hsiao Y, Mulligan DP, Nikoleris N, Petri G, Trippel C (2021) Synthesizing formal models of hardware from RTL for efficient verification of memory model implementations. In: MICRO-54: 54th annual IEEE/ACM international symposium on microarchitecture, pp 679–694

Huang B-Y, Zhang H, Subramanyan P, Vizel Y, Gupta A, Malik S (2018) Instruction-level abstraction (ILA) a uniform specification for system-on-chip (SOC) verification. ACM Trans Des Autom Electron Syst (TODAES) 24(1):1–24

Hu W, Ardeshiricham A, Kastner R (2021) Hardware information flow tracking. ACM Comput Surv (CSUR) 54(4):1–39

IEEE standard for system, software, and hardware verification and validation (2017) IEEE Std 1012-2016 (Revision of IEEE Std 1012-2012/Incorporates IEEE Std 1012-2016/Cor1-2017), pp 1–260

Kocher P, Horn J, Fogh A, Genkin D, Gruss D, Haas W, Hamburg M, Lipp M, Mangard S, Prescher T et al (2019) Spectre attacks: exploiting speculative execution. In: 2019 IEEE symposium on security and privacy (SP). IEEE, pp 1–19

Laeufer K, Koenig J, Kim D, Bachrach J, Sen K (2018) Rfuzz: coverage-directed fuzz testing of RTL on FPGAS. In: 2018 IEEE/ACM international conference on computer-aided design (ICCAD). IEEE, pp 1–8

Lipp M, Schwarz M, Gruss D, Prescher T, Haas W, Fogh A, Horn J, Mangard S, Kocher P, Genkin D et al (2018) Meltdown: reading kernel memory from user space. In: 27th {USENIX} security symposium ({USENIX} security 18), pp 973–990

Marques-Silva JP, Sakallah KA (1999) Grasp: a search algorithm for propositional satisfiability. IEEE Trans Comput 48(5):506–521

McLean J (1992) Proving noninterference and functional correctness using traces. J Comput Secur 1(1):37–57

McMillan KL (2003) Interpolation and sat-based model checking. In: International conference on computer aided verification. Springer, pp 1–13

Moskewicz MW, Madigan CF, Zhao Y, Zhang L, Malik S (2001) Chaff: engineering an efficient sat solver. In: Proceedings of the 38th annual design automation conference, pp 530–535

Reid A (2016) Trustworthy specifications of arm® v8-a and v8-m system level architecture. In: 2016 formal methods in computer-aided design (FMCAD). IEEE, pp 161–168

Sabelfeld A, Myers AC (2003) Language-based information-flow security. IEEE J Sel Areas Commun 21(1):5–19

Terauchi T, Aiken A (2005) Secure information flow as a safety problem. In: International static analysis symposium. Springer, pp 352–367

Trippel C, Manerkar YA, Lustig D, Pellauer M, Martonosi M (2017) Tricheck: memory model verification at the trisection of software, hardware, and ISA. ACM SIGPLAN Not 52(4):119–133

Trippel C, Lustig D, Martonosi M (2019) Security verification via automatic hardware-aware exploit synthesis: the checkmate approach. IEEE Micro 39(3):84–93

Volpano D, Irvine C, Smith G (1996) A sound type system for secure flow analysis. J Comput Secur 4(2–3):167–187

Wang L-T, Wu C-W, Wen X (2006) VLSI test principles and architectures: design for testability. Elsevier

Zhang H, Trippel C, Manerkar YA, Gupta A, Martonosi M, Malik S (2018) ILA-MCM: integrating memory consistency models with instruction-level abstractions for heterogeneous system-on-chip verification. In: 2018 formal methods in computer aided design (FMCAD). IEEE, pp 1–10

Bit-Level Model Checking

34

Alexander Ivrii and Yakir Vizel

Contents

Introduction 1204
Preliminaries 1205
Explicit Example: A Simple Counter 1205
Linear Time Temporal Logic 1207
Representing Systems Symbolically 1208
Algorithms for Safety Properties 1212
The Induction Principle 1212
Overview of Model Checking Algorithms 1214
Symbolic Model Checking (with BDDs) 1217
Bounded Model Checking 1218
k-Induction 1219
Interpolation and Model Checking 1221
Property Directed Reachability 1225
Combining Interpolation and PDR 1228
Summary 1229
Algorithms for Liveness Properties 1229
Introduction 1229
Overview of Model Checking Algorithms 1230
Symbolic Model Checking with BDDs 1230
Liveness-to-Safety Conversion (L2S) 1231
Bounded Liveness Checking 1232
FAIR 1234
Summary 1235

A. Ivrii (✉)
IBM, Haifa, Israel
e-mail: ALEXI@il.ibm.com

Y. Vizel (✉)
Technion - Israel Institute of Technology, Haifa, Israel
e-mail: yvizel@cs.technion.ac.il

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_35

Design Simplification Techniques 1235
Reductions 1236
Over-approximations 1237
Summary 1239
Conclusion 1239
References 1239

Abstract

Ensuring that a design conforms to its specification is an indispensable part of the modern design automation flow. *Model checking* is an automated verification technique for checking whether a given system satisfies a desired property. This problem has received much attention in the theoretical and practical domains from both industry and academia.

In this chapter, we describe some of the most important contributions made to bit-level model checking, which made it an essential tool used by hardware design companies during the development process of modern hardware designs.

Keywords

Formal verification · Bit-Level model checking · Safety · Liveness · Abstraction refinement

Introduction

The ever-increasing demand for high-performance and energy-efficient chip designs has led the industry to develop some of the most complex central processing units (CPUs), graphics processing units (GPUs) and systems on a chip (SoCs). This in turn requires an efficient verification methodology to ensure that these highly complex and sophisticated micro-architectures conform to their specifications.

Model checking (Clarke et al. 1986, 2001; Queille and Sifakis 1982) is an automated verification technique for checking whether a given system satisfies a desired property. The system is usually described as a circuit or as a finite-state machine, and the specification is given as a temporal logic formula. Unlike testing- or simulation-based verification, model checking tools are exhaustive in the sense that they examine *all* behaviors of the system and either find an erroneous behavior or confirm that the system behaves correctly. However, model checking is limited in the size of designs it can analyze. This limitation arises due to the huge state space of real-life systems and is known as the *state explosion problem*. Much of the research in this area is dedicated to increasing model checking applicability and scalability.

This chapter describes various components of modern bit-level model checking tools that are considered important for overall scalability. These include some of the most influential algorithms for verifying safety and liveness properties, as well as some of the most commonly used design simplification and abstraction

techniques. For safety verification, this presentation mainly focuses on SAT-based model checking algorithms that use the principle of inductive reasoning. The exposition is very far from being complete and should be considered only as a glimpse into the research on model checking. For instance, symbolic model checking with BDDs is described very briefly; there are many other works related to bit-level safety model checking that are not covered in this chapter; there are model checking algorithms that work at a higher level of abstraction (e.g., word-level model checking), algorithms that use machine learning, algorithms for proving equivalence, algorithms for synthesizing circuits from a high-level specification (e.g., from SystemC), and more. These, unfortunately, are outside the scope of this chapter. Nevertheless, the authors hope that this chapter portrays the wealth of research in the area, explains some of the most compelling ideas, and possibly encourages new researchers to the field.

Preliminaries

Explicit Example: A Simple Counter

Consider the hardware design that appears in Fig. 1. The Verilog code and the corresponding circuit appear in Fig. 1a and b, respectively. The design is a simple 4-bit counter c. Initially the counter is at 0, and on every step it is either nondeterministically reset back to 0 (if the reset signal rst is on) or increased by 1. Furthermore, the counter is always reset back to 0 upon reaching 8.

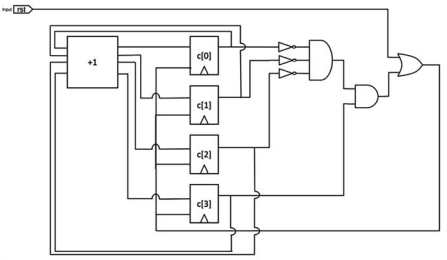
Our design can be also described by a *state machine* M that appears in Fig. 2. Since the design has 4 state elements $c[3:0]$, 4 bits are used to describe the different states in M, resulting in a total of $2^4 = 16$ states. Each single execution step of the circuit in Fig. 1b is captured by an edge in the state machine. For instance, the state 0010 has a transition to the state 0011 corresponding to increasing the value of the counter when $rst = 0$ and a transition to the state 0000 corresponding to resetting

```
input rst;
reg [3:0] c = 0;

always
if (c == 8 || rst)
      c <= 0;
else
      c <= c + 1;
end

P1: assert property (c != 6);
P2: assert property (c < 10);
P3: assert property (s_eventually (c == 8));
P4: assert property ((c == 3) |-> s_eventually (c == 0));
```

(a) Verilog code



(b) Bit-level circuit

Fig. 1 A 4-bit counter. The counter counts to 8 and resets. The counter also resets if the reset signal rst is on (**a**) Verilog code. (**b**) Bit-level circuit

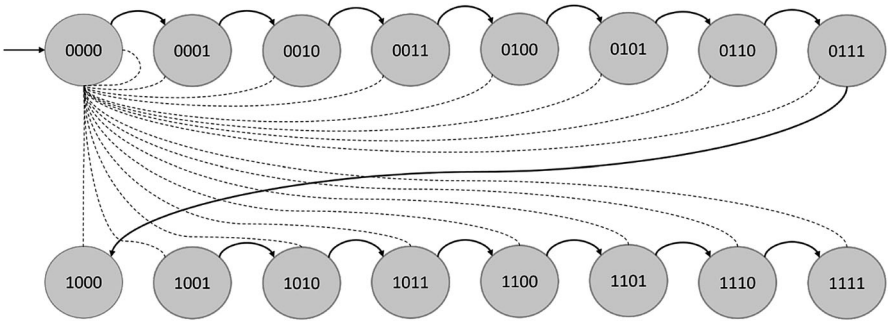


Fig. 2 A finite-state machine describing the design in Fig. 1. The state 0000 is the initial state. For clarity, the transitions to the initial state are shown as dashed. The states 1001, 1010, 1011, 1100, 1101, 1110, and 1111 cannot be reached from the initial state

the counter when $rst = 1$. The initial state $c = 0$ of the circuit translates to the state 0000 being the only initial state of M.

The Verilog code also includes several properties that one may want to check about the design. The first property $P1$ asserts that the counter never reaches the value of 6 for any possible execution of the system. Clearly, this property does not hold, as the counter will reach the value of 6 in exactly 6 steps if the reset is not triggered. Equivalently, there is a "counterexample path" of length 6 in the state machine, starting from the initial state 0000 and ending at the "bad" state 0110 violating the property:

$$0000 \to 0001 \to 0010 \to 0011 \to 0100 \to 0101 \to 0110.$$

This is not the only counterexample; for instance,

$$0000 \to 0001 \to 0000 \to 0001 \to 0010 \to 0011 \to 0100 \to 0101 \to 0110$$

is another counterexample, with the transition from 0001 to 0000 due to reset occurring on the second cycle. However, one can easily check that there are no counterexample paths of length smaller than 6, that is, the initial state 0000 cannot reach the bad state 0110 in a sequence of 5 steps or less. The second property $P2$ asserts that the counter is always smaller than 10 for any possible execution of the system. Clearly, this property holds. In fact, the set of states reachable from the initial state is given by

$$\{0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000\},$$

and each of these states satisfies the condition $c < 10$. The properties $P1$ and $P2$ are the so-called *safety properties* that (intuitively) say that "some bad thing never happens."

The property $P3$ asserts that for any possible execution of the system, starting from any state in the execution, the counter will always eventually reach the value of 8. This property does not hold: consider the execution where the reset signal is triggered on every second step; the counter will never reach the value of 3, let alone 8. However, an *infinite counterexample* is needed to explain the failure; no finite-length explanation will be sufficient. In fact, every finite-length execution can be completed to an execution where the counter does reach the value of 8. Finally, the property $P4$ asserts that for any possible execution of the system, starting from the state where $c = 3$, the counter must eventually reset back to 0. This property holds: either the reset signal occurs and the counter is reset, or the reset signal does not occur and the counter eventually reaches 8 and then becomes 0 on the next cycle. The properties $P3$ and $P4$ are the so-called *liveness properties* that (intuitively) say that "some good thing must keep happening."

Thinking in terms of states of the finite-state machine M gives a natural way to analyze the design. For example, in order to check that the system satisfies a given safety property, one only needs to make sure that the property holds on every state that can be reached from the initial state. Any graph search algorithm, such as *breadth-first search* or *depth-first search*, would provide an answer. However, a fundamental problem with this approach is the so-called *state explosion problem*. The current design has 4 state elements and requires $2^4 = 16$ to explicitly represent the states in the corresponding finite-state machine. In general, a design with n state elements requires 2^n states. Unfortunately, the approach of explicitly constructing all possible states only works for very small values of n.

Linear Time Temporal Logic

More general properties (specifications) of the design can be described using *temporal logic*. This chapter considers the linear time temporal logic (LTL) (Pnueli 1977). LTL uses temporal operators G, F, X, and U, which stand for "globally," "eventually," "next," and "until." Common examples of LTL properties used in the industry include:

- Mutual exclusion (i.e., two signals cannot be 1 at the same time): $G(\neg sig_1 \vee \neg sig_2)$
- Every computation is completed within three steps: $G(started \rightarrow (ended \vee Xended \vee XXended \vee XXXended)$
- Every request is eventually granted: $G(req \rightarrow Fgrant)$
- If a request arrives, it must be granted before any other request arrives: $G(req \rightarrow X(\neg reqUgrant))$

Using the automata-theoretic construction (Vardi 2007), a general safety property can be expressed in the form Gp, where p is a signal (Boolean variable) that represents the "good states." Similarly, a general liveness property can be expressed

in either of the forms GFp or FGq, for suitable signals p and q. See also Rozier (2011) for a recent survey on the matter. In what follows, the considered safety/liveness properties will always be in one of the forms above.

Representing Systems Symbolically

To cope with the state explosion problem, modern model checking algorithm represents states *implicitly* using formulas and *approximates* the set of reachable states.

Transition System This section explains how it is possible to reason about states in the design using formulas. Let V be a set of Boolean variables representing state elements in the design. The set S of all states can be described as $S \stackrel{\text{def}}{=} \mathbb{B}^{|V|}$, where $\mathbb{B} := \{0, 1\}$. Given a formula F over V, one can consider F as representing the set of states in which it is satisfiable, that is, $\{s \in S \mid F(s) = 1\}$. Conversely, a set of states can be described by any formula that represents it. In this case, it is often said that the set of states is described *implicitly* or *symbolically*. In particular, a single state $s \in S$ can be represented as a conjunction of literals (which are satisfied in s). This already allows to represent the initial states of the design using a formula $Init(V)$ and (for safety model checking) represent the "good" states of the design using a formula $P(V)$. To describe transitions between states, reasoning about *pairs* (s, s') of states is required, with s representing the starting state and s' representing the successor state, respectively. It is also common to refer to s as the "current state" and to s' as the "next state" of a transition. The set of all possible transitions can be represented by a formula $Tr(V, V')$ over V and V', called the *transition relation*. The non-primed variables are usually referred to as *current state variables*, and the primed variables are referred to as *next-state variables*.

The above discussion can be formalized as follows:

Definition 1. A *finite-state transition system* is a tuple $M \stackrel{\text{def}}{=} \langle V, Init, Tr \rangle$, where V is a set of Boolean variables, $Init(V)$ is a formula over V describing the initial states, and $Tr(V, V')$ is a formula over the variables V and their primed counterparts $V' = \{v' \mid v \in V\}$ describing transitions between states.

Example 1. Consider the state machine that appears in Fig. 2. Let us denote the set of variables for M as $V = \{v_0, v_1, v_2, v_3\}$, where v_i describes the ith bit of c. The initial states are described by the formula

$$Init(V) := \neg v_3 \wedge \neg v_2 \wedge \neg v_1 \wedge \neg v_0.$$

The safety property $P2$ asserts that $c < 10$ and can be represented as

$$P(V) := (\neg v_3) \vee (v_3 \wedge \neg v_2 \wedge \neg v_1).$$

The transition relation is a bit more complex due to many transitions between states. A partial definition containing only outgoing edges from 0000 and 0001 appears below:

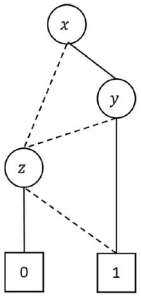
$$\begin{aligned} Tr(V, V') := &(\neg v_3 \wedge \neg v_2 \wedge \neg v_1 \wedge \neg v_0 \wedge \neg v'_3 \wedge \neg v'_2 \wedge \neg v'_1 \wedge \neg v'_0) \vee \\ &(\neg v_3 \wedge \neg v_2 \wedge \neg v_1 \wedge \neg v_0 \wedge \neg v'_3 \wedge \neg v'_2 \wedge \neg v'_1 \wedge v'_0) \vee \\ &(\neg v_3 \wedge \neg v_2 \wedge \neg v_1 \wedge v_0 \wedge \neg v'_3 \wedge \neg v'_2 \wedge \neg v'_1 \wedge \neg v'_0) \vee \\ &(\neg v_3 \wedge \neg v_2 \wedge \neg v_1 \wedge v_0 \wedge \neg v'_3 \wedge \neg v'_2 \wedge v'_1 \wedge \neg v'_0) \vee \ldots \end{aligned}$$

BDDs and SAT Representing sets of states using logical formulas would not be very useful if there was no way to represent the formulas concisely or to efficiently reason about them. Fortunately, such ways exist, with two predominant approaches: an approach based on *binary decision diagrams* (BDDs) and an approach based on *satisfiability solving* (SAT).

A binary decision diagram is a data structure that concisely represents Boolean functions using a directed, acyclic graph (Bryant 1986). This data structure is very versatile, allowing to compute complementation, conjunction, disjunction, and even existential and universal quantification. Furthermore, (reduced ordered) BDDs are canonical (given variable order), thus allowing an efficient check whether two functions are equivalent. BDD-based model checking (Burch et al. 1990) constituted the first breakthrough of relative scalability, scaling to designs with up to several hundred state elements. Unfortunately, BDDs are not very efficient beyond this point.

The Boolean satisfiability problem (SAT) is the problem of determining whether the variables of a Boolean formula can be assigned to *true* or *false* in such a way that the formula evaluates to *true*. If this is the case, the formula is called *satisfiable*; otherwise, it is called *unsatisfiable*. In practice, Boolean formulas are often presented in *conjunctive normal form* (CNF): a CNF is a conjunction of *clauses*, where each clause is a disjunction of *literals*, and each literal is either a variable or its negation. General Boolean functions can be translated to CNF using the Tseitin transformation (Tseitin 1983), with the idea of introducing an additional variable for each logical operator in the formula. While SAT is a NP-complete problem (Cook 1971), due to many advances in the field, modern SAT algorithms (SAT solvers) are able to solve formulas involving tens of thousands of variables and millions of clauses.

Example 2. Figure 3 shows the function $f = (x \wedge y) \vee \neg z$ represented both as BDD and in CNF. For the CNF representation, Tseitin transformation is used, with an additional variable a representing $x \wedge y$. In this case, a simpler CNF representation without auxiliary variables is also possible: $f = (x \vee \neg z) \wedge (y \vee \neg z)$.



$(\neg a \vee x)$
$(\neg a \vee y)$
$(a \vee \neg x \vee \neg y)$
$(a \vee \neg z)$

(a) BDD representation (b) CNF representation

Fig. 3 Boolean function $(x \wedge y) \vee \neg z$ represented as a binary decision diagram (left), and in conjunctive normal form, using an auxiliary variable a (right) (**a**) BDD representation. (**b**) CNF representation

Notation Given two formulas F and G over V, the notation $F \Rightarrow G$ means that F logically implies G (i.e., every assignment which satisfies F also satisfies G). For instance, given the formulas in Example 1, $Init \Rightarrow P$ means that the property P holds on every initial state.

Given a formula F over V, the primed formula F' denotes the corresponding formula in which all variables $v \in V$ have been replaced with their primed versions $v' \in V'$. In the context of multiple steps of the transition system, the notation $V^i \stackrel{\text{def}}{=} \{v^i | v \in V\}$ is used instead of V' to denote the variables in V after i steps. Given a formula F over V^i, the formula $F[V^i \leftarrow V^j]$ is identical to F except that for every variable $v \in V$ each occurrence of v^i in F is replaced with v^j. This substitution allows us to change the execution step to which a formula refers.

Substitution is also used for formulas and sub-formulas. Let $F(V)$ and $H(V)$ be formulas over V and let $G(V)$ be a sub-formula of F. $F[G \leftarrow H]$ denotes the formula obtained by replacing all occurrences of the sub-formula G in F with H.

Symbolic Successor Computation Let M be a transition system and let F be a formula over V describing a set of states. Suppose that the goal is to find all successors of F in M. If there is an explicit graph for M, it suffices to iterate overall outgoing edges of states that are in F and by that compute the set of successors. This is not possible when M is given symbolically. In order to compute the set of successors symbolically, the *post-image* operator is defined as follows:

$$PostImg(F, Tr) := (\exists V \cdot F(V) \wedge Tr(V, V'))[V' \leftarrow V]$$

$PostImg(F, Tr)$ is a formula over V describing all successors of F. First, note that $F(V) \wedge Tr(V, V')$ is a formula over $V \cup V'$. Let ϕ be a satisfying assignment of $F(V) \wedge Tr(V, V')$, and let us denote the projection of ϕ onto V as $s(V)$ and its projection onto V' as $t(V')$. Hence, ϕ represents the pair $(s, t) \in Tr$ such that s is a

state in F. It is easy to conclude that all satisfying assignments to $F(V) \wedge Tr(V, V')$, when projected onto V', represent all successors of F in M.

Example 3. Let's compute $PostImg(Init, Tr)$ in our running example. It follows that:

$$Init(V) \wedge Tr(V, V') = ((\neg v_3 \wedge \neg v_2 \wedge \neg v_1 \wedge \neg v_0 \wedge \neg v'_3 \wedge \neg v'_2 \wedge \neg v'_1 \wedge \neg v'_0) \vee \\ \vee (\neg v_3 \wedge \neg v_2 \wedge \neg v_1 \wedge \neg v_0 \wedge \neg v'_3 \wedge \neg v'_2 \wedge \neg v'_1 \wedge v'_0)),$$

where the first disjunct corresponds to the transition from 0000 back to 0000 and the second disjunct corresponds to the transition from 0000 to 0001. All the other disjuncts in $Tr(V, V')$ disappear as they are incompatible with $Init(V) = \neg v_3 \wedge \neg v_2 \wedge \neg v_1 \wedge \neg v_0$. Thus,

$$\exists v_0, v_1, v_2, v_3 \cdot (Init(V) \wedge Tr(V, V')) \\ = (\neg v'_3 \wedge \neg v'_2 \wedge \neg v'_1 \wedge \neg v'_0) \vee (\neg v'_3 \wedge \neg v'_2 \wedge \neg v'_1 \wedge v'_0),$$

and

$$PostImg(Init, Tr) \equiv (\neg v_3 \wedge \neg v_2 \wedge \neg v_1 \wedge \neg v_0) \vee (\neg v_3 \wedge \neg v_2 \wedge \neg v_1 \wedge v_0).$$

The above formula can be written even more compactly as $PostImg(Init, Tr) \equiv \neg v_3 \wedge \neg v_2 \wedge \neg v_1$.

Computing Reachable States The post-image operator gives us a way to traverse the state space of M, starting from any set of states, and particularly the initial states. This allows to define that a state s is reachable in M iff s belongs to the transitive closure of $PostImg(Init, Tr)$. Equivalently, a state s in M is reachable if there exists a path that starts in an initial state and ends with s.

Similarly to how $PostImg$ is defined, a path of length k in a transition system $M = \langle V, Init, Tr \rangle$ is described by the following formula:

Formula 1. $path^{i,j} \stackrel{\text{def}}{=} Tr(V^i, V^{i+1}) \wedge \ldots \wedge Tr(V^{j-1}, V^j)$

where $0 \leq i \leq j$ and $j - i = k$. Note that $path^{i,j} \equiv \top$ (viz., $True$) when $i = j$. An *initial* path of length k is defined using the formula $Init(V^0) \wedge path^{0,k}$.

The notation introduced above illustrates how a traversal of the state space of a transition system M can be reduced to satisfiability. In essence, an initial *path* formula of a transition system describes all possible executions of length k of the circuit when starting from the initial condition. Thus, it also represents all states that can be reached in k transitions of M.

Algorithms for Safety Properties

Model checking algorithms aim to establish the safety of a given transition system or provide a counterexample if the system is not safe. A model checking algorithm is *complete* if it is able to either provide a counterexample or prove the absence of counterexamples of any length and *sound* if it does not provide a wrong answer.

The Induction Principle

Most model checking algorithms are based on some form of inductive reasoning. Consider a transition system $M = \langle V, Init, Tr \rangle$ and a (safety) property P.

First, let's suppose that every initial state of M satisfies P (i.e., $Init \Rightarrow P$) and that every state that satisfies P can only transition to another state that satisfies P (i.e., $P \wedge Tr \Rightarrow P'$). It follows then that all the reachable states in M must satisfy P. This type of reasoning is called *the principle of induction*, and the property P is said to be *inductive*. Equivalently, if as a set of states P includes all of the initial states and is *closed* under the transition relation, then it necessarily contains all of the reachable states (and, in addition, it may also include some of the non-reachable states). Checking whether P is inductive can easily be done using a SAT solver, by checking whether both formulas $Init \wedge \neg P$ and $P \wedge Tr \wedge \neg P'$ are unsatisfiable.

In practice, however, the property P is usually not closed under the transition relation (and thus is not inductive). In this case, $P \wedge Tr \Rightarrow P'$ does not hold, or equivalently the formula $P \wedge Tr \wedge \neg P'$ is satisfiable. It follows that there exist states s and t, such that $(s, t) \in Tr$, $s \in P$, and $t \notin P$. The state s is usually referred to as a *counterexample to induction*. This does not mean that M is not safe with respect to P; it only means that the safety of P cannot be proved by the principle of induction alone. So, in order to show that P holds in all reachable states, model checking algorithms use a more complex type of inductive reasoning in the form of a *safe inductive invariant*:

Definition 2 (Inductive Invariant). A set of states characterized by the formula F is *inductive* if $F \wedge Tr \Rightarrow F'$ holds. F is inductive *relative* to a formula G if $G \wedge F \wedge Tr \Rightarrow F'$ holds. F satisfies *initiation* if $Init \Rightarrow F$, i.e., if F comprises all initial states. F is *safe* with respect to P if $F \Rightarrow P$.

A formula F is called an *inductive invariant* if it satisfies initiation and is inductive.

Lemma 1. *Let $M = \langle V, Init, Tr \rangle$ be a transition system, let P be a property, and let F be a propositional formula representing a set of states. If F is a safe inductive invariant, then P holds in M.*

The above lemma leads to the perspective that the goal of a model checking algorithm is to either synthesize a safe inductive invariant (and by that to prove that

the property holds) or to prove that such an inductive invariant does not exist (usually by finding a counterexample). Checking whether F is a safe inductive invariant for P can be easily done using a SAT solver, by checking whether the formulas $Init \wedge \neg F$, $F \wedge Tr \wedge \neg F'$, and $F \wedge \neg P$ are all unsatisfiable. However, finding an inductive invariant (when it exists) is a very challenging task. Model checking algorithms approach this task by traversing the state space and constructing an *inductive trace*:

Definition 3. An *inductive trace* of length k with respect to a transition system $M = \langle V, Init, Tr \rangle$, denoted $\vec{F}_{[k]}$, is a sequence $\langle F_0, \ldots, F_k \rangle$ of propositional formulas over V such that the following holds:

- $F_0 = Init$
- $F_i \wedge Tr \Rightarrow F'_{i+1}$ for $0 \leq i < k$

An inductive trace $\vec{F}_{[k]}$ is *monotonic* if $F_i \Rightarrow F_{i+1}$ for $0 \leq i < k$ and is *safe* w.r.t. a property P if $F_i \Rightarrow P$ for $0 \leq i \leq k$. The individual propositional formulas F_i in a trace $\vec{F}_{[k]}$ are called *frames* of the trace.

In what follows, an inductive trace $\vec{F}_{[k]}$ is referred to as a *trace*, and the subscript $[k]$ is dropped when clear from the context. In addition, if $\vec{F}$ is safe w.r.t. a property P and when P is clear from the context, $\vec{F}$ is referred to as safe. An element F_i in a trace $\vec{F}_{[k]}$ represents an over-approximation of all states reachable in i steps of the transition system. If the trace is monotonic, then F_i is an over-approximation of all states reachable in at most i steps. Monotonic traces arise (1) in the context of BDD-based model checking (Burch et al. 1990), where the set of reachable states is iteratively increased until either a fixed point is reached or a counterexample is detected, and (2) in SAT-based model checking algorithms such as ISB (Vizel and Grumberg 2009), PDR (Bradley 2011), and AVY (Vizel and Gurfinkel 2014).

The following definition and lemma highlight the relationship between an inductive invariant (Definition 2) and an inductive trace (Definition 3).

Definition 4. Let $\vec{F}_{[k]}$ be a trace. Let us define $F^i \stackrel{\text{def}}{=} \bigvee_{j=0}^{i} F_j$ where $0 \leq i < k$. If there exists $0 \leq i < k$ such that $F_{i+1} \Rightarrow F^i$, then $\vec{F}$ is referred to as closed w.r.t. i.

Lemma 2. *Let $M = \langle V, Init, Tr \rangle$ be a transition system, let P be a property, and let $\vec{F}_{[k]}$ be a trace. If $\vec{F}$ is closed w.r.t. $0 \leq i < k$, then F^i is an inductive invariant. In addition, if $\vec{F}_{[k]}$ is* safe *w.r.t. P, then $F^i \Rightarrow P$ holds, and thus P is safe.*

To summarize, most model checking algorithms construct inductive traces of length k for increasing values of k. Whenever an inductive trace becomes closed (with respect to some i), the algorithm discovers an inductive invariant. If, in addition, the invariant happens to be safe (with respect to P), then P holds on all reachable states. The details of the algorithms (and various heuristics) differ between the algorithms and will be discussed in the following sections.

Overview of Model Checking Algorithms

This section provides a brief overview of the principles and mechanics underlying a number of widely used model checking techniques for safety properties. These techniques are summarized in Table 1, illustrated in Figs. 5, 6, 7, and 8, and described in detail in sections "Symbolic Model Checking (with BDDs)" to "Combining Interpolation and PDR ". Whenever possible, the techniques are illustrated using the transition system $M = \langle V, Init, Tr \rangle$ described in section "Explicit Example: A Simple Counter", except that the reset signal rst is assumed to be always off. This allows to write the transition relation in a slightly simplified form as $Tr = (c' = (c == 8)\ ?\ 0 : c + 1)$. The safety property considered is $P : c < 10$.

Symbolic Model Checking (SMC) with BDDs Using logical formulas to traverse the state space of a given transition system and representing them using reduced ordered binary decision diagrams (ROBDDs, BDDs for short) (Bryant 1986) was introduced in the seminal work of McMillan et al. (Burch et al. 1990). In fact, this work has enabled the application of model checking algorithms to realistic circuits with hundreds of state variables. SMC uses the post-image operator, which for a given set of states computes all states that are reachable from that set of states in

Table 1 Summary of safety model checking algorithms

Algorithms	Details	Summary	Refs
SMC	Section "Symbolic Model Checking (with BDDs)"	Symbolic model checking using BDDs. Efficient on small designs. Does not scale to large designs, due to the size of BDDs	Burch et al. (1990)
BMC	Section "Bounded Model Checking"	SAT-based bounded model checking. Very efficient for finding counterexamples. Cannot prove properties	Biere et al. (1999)
k-induction	Section "k-Induction"	Extension of BMC that can prove properties. Uses generalized principle of k-induction. Occasionally efficient in practice	Sheeran et al. (2000)
ISB, ITP	Section "Interpolation and Model Checking"	Interpolation-based model checking. Uses monolithic SAT queries. Quite efficient in practice	Vizel and Grumberg (2009); McMillan (2003)
IC3, PDR	Section "Property Directed Reachability"	Property-driven reachability. Does not unroll the transition system and uses incremental single-step SAT queries. Very efficient in practice	Bradley (2011); Een et al. (2011)

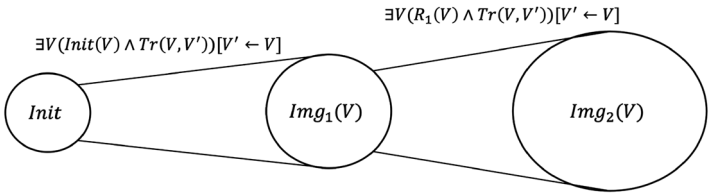


Fig. 4 The post-image operator

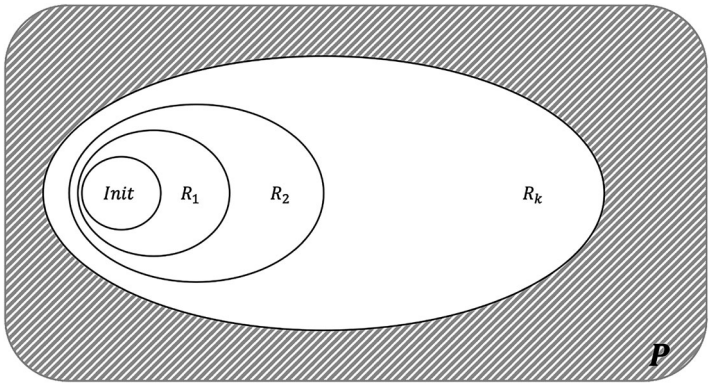


Fig. 5 Symbolic model checking with BDDs. The set R_i represents all states that are reachable in i steps or less from the initial states. Each R_i is obtained as the disjunction of R_{i-1} and the post-image operator applied to R_{i-1}. Note that $R_0 := Init$

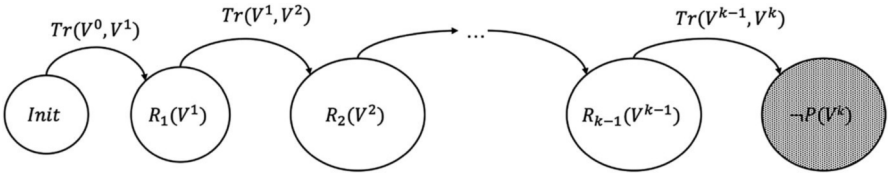


Fig. 6 *Bounded model checking* (BMC) checks whether a property P can be violated in k steps by encoding reachable sets of states $(R_1, \ldots, R_k)$ as a SAT instance (without computing R_is explicitly). BMC does not identify repeatedly visited states and cannot determine whether the property holds for arbitrarily many steps

one step of the transition relation Tr. The post-image operator is used iteratively, starting from the initial states in order to compute all states that are reachable from the initial states. This process continues until either it discovers a bad state that is reachable or it reaches a fixed point, concluding that no reachable state is a bad state. In the former case, a counterexample is found, while, in the latter case, the property is proved correct. The iterative application of the post-image operator is illustrated in Figs. 4 and 5.

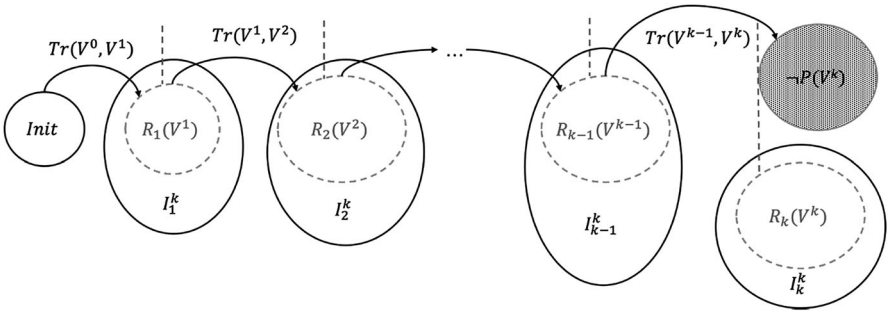


Fig. 7 *Interpolation sequence-based* model checking partitions an unsatisfiable BMC instance φ^k into $k+1$ parts (with the last part representing the "bad" states), resulting in a sequence interpolant $\langle I_1^k, \ldots I_k^k \rangle$. Each I_i^k over-approximates states reachable in i steps; moreover, states in I_{i+1}^k over-approximate states reachable from I_i^k in a single step

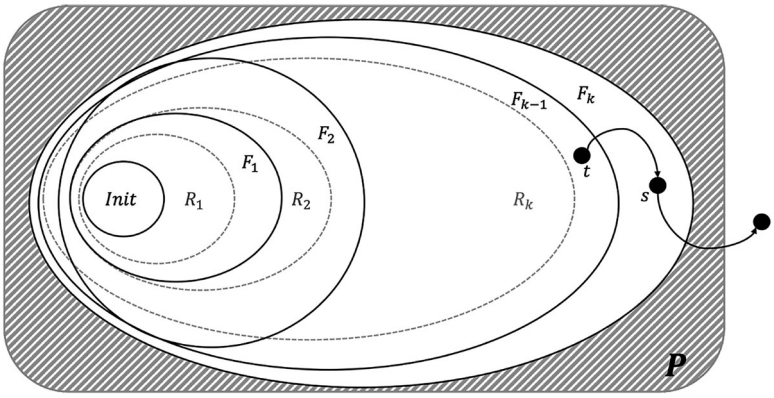


Fig. 8 PDR maintains a *monotonic* sequence of frames $F_1, \ldots, F_k$ which over-approximate the states reachable in up to k steps. The approximation is iteratively refined by eliminating from frame F_k states s that can reach $\neg P$ but have themselves no predecessor in F_{k-1} (i.e., $\neg s$ is inductive relative to F_{k-1} and $F_{k-1} \wedge \neg s \wedge Tr \Rightarrow \neg s'$ holds)

Bounded Model Checking (BMC) The success of BMC (Biere et al. 1999) is based on its ability to find counterexamples. BMC is based on the exploration of bounded paths in a transition system M. To this end, BMC *unwinds* the transition relation Tr, as illustrated in Fig. 6 and explained in section "Bounded Model Checking", in order to determine whether the property P can be violated in exactly k steps.

k-Induction k-Induction (Sheeran et al. 2000) is a generalization of the principle of induction. It aims to find a bound $k \geq 1$ for which the following statements are true: (the "base case") P holds on all the states reachable in k steps or less from the

initial states, and (the "step case") whenever P holds for k consecutive steps of the transition system, then it necessarily also holds in the subsequent step. One can see that if such a k exists, then P holds on all reachable states and that the principle of induction corresponds to the case of $k = 1$. k-Induction is usually used with BMC. With the additional *unique state constraints*, k-induction is complete (Sheeran et al. 2000), namely, if a transition system M is safe, then there exists a k for which both the base and step case can be proved.

Interpolation-Based Model Checking Interpolation-based model checking algorithms (McMillan 2003) also explore bounded paths in M but use *interpolation* to synthesize an inductive invariant during the exploration. For a pair of formulas $\alpha(X, Y)$ and $\beta(Y, Z)$, if $\alpha \wedge \beta$ is unsatisfiable, the *interpolation theorem* states that there exists a formula $\gamma(Y)$ such that $\alpha \Rightarrow \gamma$ and $\gamma \wedge \beta$ is unsatisfiable. Intuitively, one can think of γ as an over-approximation of existential quantifier or, in the context of model checking, an over-approximation of the post-image operator (see Section "Interpolation and Model Checking").

As illustrated in Fig. 7, interpolants I_i^k derived from unsatisfiable BMC instances safely over-approximate reachable states. The resulting formulas I_i^k are then incorporated into a *safe* trace which is maintained by the model checker and gradually refined until either an inductive invariant or a counterexample is found. The strength of interpolation-based model checking is in its ability to compute concise over-approximations of reachable states, thus accelerating fixed-point convergence.

Property Directed Reachability The PDR algorithm (Bradley 2011; Een et al. 2011) (originally called IC3 (IC3 stands for "Incremental Construction of Inductive Clauses for Indubitable Correctness"; PDR stands for " Property Directed Reachability.")) differs from the abovementioned algorithms as it does not explicitly unroll the transition relation (viz., it does not use the $path^{i,j}$ formulas). PDR maintains a *monotonic safe* trace (as illustrated in Fig. 8), which is incrementally refined by eliminating states that can be proven unreachable by means of consecution checks (Definition 2) over subsequent frames. PDR's focus on single steps of the transition relation enables an efficient and targeted generation of relatively inductive clauses. Unlike interpolation-based techniques, PDR does not depend on the (usually unpredictable) results produced by an interpolation engine.

Symbolic Model Checking (with BDDs)

Given a transition system M, model checking algorithms are based on an exhaustive exploration of the state space of M, either proving that the property under verification holds in every reachable state or finding a counterexample in the form of a path from the initial states to a state that violates the property.

In the early days of model checking, the algorithms were "explicit" in the sense that they operated on an explicit-state graph representing M and traversed its states

one by one. This limited the applicability of model checking algorithms for two obvious reasons: this requires to (i) construct the state graph and (ii) traverse its states. Achieving both of these is intractable for state graphs of realistic systems, simply because they consist of too many states. For example, a hardware design with a 32-bit register has 2^{32} states. As a result, explicit model checking algorithms could not be applied to realistic systems.

A decade after, in the seminal work of McMillan et al. (Burch et al. 1990), the paradigm has changed with the introduction of symbolic model checking (SMC) using binary decision diagrams (BDDs). Unlike explicit model checking algorithms, SMC does not require to construct the state graph and does not explicitly traverse the states of the system one by one. Instead, it uses logical formulas to represent the system and various sets of states. These logical formulas are represented using a data structure called BDDs (Bryant 1986) (The interested reader can find all details about BDDs and their use in this context in Clarke et al. 2001.). This new paradigm allowed to apply model checking to systems that are orders of magnitude larger than what was possible with explicit model checking.

BDD-based symbolic model checking is an iterative process for checking P in all states reachable from $Init$ in M. At each iteration i, SMC computes the set R_i of states that are reachable in i steps or less from the initial states and checks if all the states in R_i satisfy P. The computation uses the post-image operator (see Section "Representing Systems Symbolically" and Figs. 4 and 5). Specifically, R_is are defined in the following manner:

$$R_0 := Init \qquad R_i := R_{i-1} \vee PostImg(R_{i-1}) \tag{1}$$

For each such set R_i of reachable states, SMC checks whether $R_i \Rightarrow P$, that is, whether all the states are reachable in i steps or less satisfy the property P. If this implication does not hold, then there exists a state that is reachable in at most i steps that violates P. Hence, SMC concludes that a counterexample is found.

If the implication does hold, SMC checks if a fixed point is reached. Note that from the definition of R_i, it holds that $R_{i-1} \Rightarrow R_i$. Therefore, a fixed point is reached when $R_i \Rightarrow R_{i-1}$. In case a fixed point is found, SMC concludes that all reachable states satisfy P and hence the property holds. Otherwise, it moves to the next iteration. Note that $\vec{F} := \langle R_0, R_1, \ldots, R_k \rangle$ is a trace as per Definition 3.

Bounded Model Checking

BMC (Biere et al. 1999) is an iterative process for checking P on all initial paths of M up to a given bound on the length. BMC is a SAT-based algorithm. Given a transition system M and a specific length k, BMC translates the question "Does M have a counterexample of length k?" into a propositional formula and uses a SAT solver to determine if the formula is satisfiable or not. As explained in Section "Representing Systems Symbolically", a SAT solver can either find a satisfying assignment or prove its absence. If the solver finds a satisfying

```
1  Function BMC(M, P, N)
2      k ← 0;
3      while k <= N do
4          if IsSAT(φ^k) then
5              return CEX;
6          end
7          k = k + 1;
8      end
9  end
10 return P holds up to N
```

Algorithm 1: Bounded model checking for a transition system M up to bound N

assignment, a counterexample exists and is represented by the assignment. If the solver proves that no satisfying assignment exists, then BMC concludes that no counterexample of length k exists.

The pseudocode for BMC appears in Algorithm 1. In order to conclude that there is no counterexample of length N or less, BMC iterates overall lengths from 0 up to the given bound N. For a given value of k, the following propositional formula is built and passed to a SAT solver:

Formula 2.

$$\begin{aligned}\varphi^k &\stackrel{\text{def}}{=} Init(V^0) \wedge path^{0,k} \wedge \neg P(V^k)\\ &= Init(V^0) \wedge Tr(V^0, V^1) \wedge \ldots \wedge Tr(V^{k-1}, V^k) \wedge \neg P(V^k)\end{aligned}$$

The formula φ^k *implicitly* represents all paths of length k in the transition system that reach a bad state at step k. If there exists a satisfying assignment for φ^k, then there exists a path of length k violating P, and the property does not hold.

In theory, BMC can conclude that the property holds on all reachable states once N exceeds the *diameter* of the transition system: the length of the longest path among all shortest paths from an initial state to some other state in the system. However, in practice, it is hard to compute this bound, and even when known, it is often too large to handle (Clarke et al. 2004). BMC cannot compute a trace and therefore cannot find an inductive invariant. Thus, in practice, the main drawback of BMC is its incompleteness: BMC can only prove the absence of counterexamples of length up to a given bound but cannot guarantee that there is no counterexample of arbitrary length.

k-Induction

As described in the previous section, in practice BMC is only able to find counterexamples. k-Induction (Sheeran et al. 2000) can be used in conjunction with BMC in order to obtain proofs as well. As mentioned in the overview, k-Induction is

a generalization of the *induction principle* and consists of two steps: (1) a base case, which checks that all initial paths of length k satisfy P, and (2) a step case (viz., the induction step), which checks that all paths of length k that satisfy P can only be extended to paths of length $k+1$ that also satisfy P. A path is said to satisfy P if every state along the path satisfies P. Note that such a path does not have to start at an initial state.

The two steps are now described in more detail:

- **Base Case:** Checking that all initial paths of length k satisfy P is equivalent to proving there is no counterexample of length k or less. Hence, this step can be executed using BMC. More precisely, by showing that φ^i is unsatisfiable for $0 \leq i \leq k$.
- **Step Case:** To prove that every path of length k that satisfies P can only be extended to a path of length $k+1$ that also satisfies P, the following formula needs to hold: $(\bigwedge_{i=0}^{k} P(V^i)) \wedge path^{0,k+1} \Rightarrow P(V^{k+1})$. To check if the above formula holds, the following formula is built:

  Formula 3. $\chi^k \stackrel{\text{def}}{=} (\bigwedge_{i=0}^{k} P(V^i)) \wedge path^{0,k+1} \wedge \neg P(V^{k+1})$

  and a SAT solver checks if it is unsatisfiable. If it is indeed unsatisfiable, k-induction concludes that the property P is $(k+1)$-inductive, and therefore P holds. Otherwise, k is incremented and the algorithm moves back to the "base case."

k-Induction is mostly used in conjunction with BMC and is executed in an iterative fashion, starting from bound 0 up to a given threshold bound N. The pseudocode appears in Algorithm 2.

```
1  Function kIND(M, P, N)
2      if IsSAT(φ^0) then
3          return CEX;
4      end
5      k ← 1;
6      while k <= N do
7          if IsSAT(φ^k) then
8              return CEX;
9          end
10         if IsUnSAT(χ^k) then
11             return P holds;
12         end
13         k = k + 1;
14     end
15 end
16 return P holds up to N
```

Algorithm 2: k-Induction for a transition system M up to bound N

Example 4. Consider the transition system M from Fig. 2 and the property $P := (c < 10)$. One can check that P is not 1-inductive. Write the formula $P \wedge Tr \wedge \neg P'$ in a simplified form:

$$(c < 10) \wedge (c' = (c == 8)\ ?\ 0 : c + 1) \wedge (c' >= 10).$$

It can be satisfied by setting $c = 9$ (and $c' = 10$).

However, one can check that P is 2-inductive. Write the formula $P \wedge Tr \wedge P' \wedge Tr \wedge \neg P''$ in a simplified form:

$$\begin{aligned}(c < 10) \wedge (c' = (c == 8)\ ?\ 0 : c + 1) \wedge \\ \wedge (c' < 10) \wedge (c'' = (c' == 8)\ ?\ 0 : c' + 1) \wedge (c'' >= 10)\end{aligned}$$

This formula is unsatisfiable. Intuitively, if $c \leq 8$, then $c'' \geq 10$ cannot hold. And if $c \geq 9$, then $c' < 10$ cannot hold. By also checking that P holds on all paths of length 0 and 1, it follows that P holds unboundedly on the design.

By further strengthening the "inductive step" χ^k formulas with the *simple path constraint* (which requires all the states on the path $path^{0,k+1}$ to be distinct), k-induction becomes a complete algorithm (Sheeran et al. 2000); that is, every true property can be proven by k-induction for a suitable value of k. However, in practice k-induction only works when a small value of k suffices for the proof, as otherwise the formulas become too difficult for a SAT solver. As it happens, in practice, k-induction is not able to solve most properties of interest, and other forms of inductive reasoning are needed.

Interpolation and Model Checking

Multiple complete SAT-based model checking algorithms are based on the concepts of *interpolation*.

Craig Interpolation. Given a pair of formulas (A, B) such that $A \wedge B$ is unsatisfiable, a *Craig interpolant* (Craig 1957) for (A, B) is a formula I such that:

$$A \Rightarrow I \qquad I \Rightarrow \neg B \qquad \mathcal{L}(I) \subseteq \mathcal{L}(A) \cap \mathcal{L}(B) \tag{2}$$

where $\mathcal{L}(A)$ denotes the set of all atomic propositions in A. Such I always exists for an unsatisfiable pair (A, B).

Sequence Interpolation. A *sequence interpolant* (Jhala and McMillan 2005) (also called an *interpolation sequence*) extends interpolation to a sequence of formulas. Let $\vec{G} = \langle G_1, \ldots, G_N \rangle$ denote a sequence with N elements and $\vec{G}_i$ denote the ith element of the sequence. Given an unsatisfiable sequence of formulas $\vec{A} :=$

$\langle A_1, \ldots, A_N \rangle$, i.e., $A_1 \wedge \cdots \wedge A_N \Rightarrow \bot$, a *sequence interpolant* $\vec{I} = \text{SEQITP}(\vec{A})$ for $\vec{A}$ is a sequence of formulas $\vec{I} = \langle I_1, \ldots, I_{N-1} \rangle$ such that

$$A_1 \Rightarrow I_1 \qquad \forall 1 < i < N \cdot I_{i-1} \wedge A_i \Rightarrow I_i \qquad I_{N-1} \wedge A_N \Rightarrow \bot \tag{3}$$

and for all $1 \leq i < N$, $\mathcal{L}(I_i) \subseteq \mathcal{L}(A_1 \wedge \cdots \wedge A_i) \cap \mathcal{L}(A_{i+1} \wedge \cdots \wedge A_N)$.

The two model algorithms described next suitably combine computation of interpolants and sequence interpolants with BMC. Historically, the work McMillan (2003) is the first work to apply interpolation in the context of model checking, using interpolants to over-approximate sets of reachable states and to compute an inductive trace. However, the work Vizel and Grumberg (2009) is described first, due to its relative simplicity.

Interpolation Sequence-Based Model Checking (ISB)

In Vizel and Grumberg (2009) an interpolation sequence-based (ISB) algorithm is suggested for the computation of a *safe* trace as part of the main BMC loop. When a BMC query is unsatisfiable, a sequence interpolant is extracted from the proof of unsatisfiability. ISB is integrated in BMC's main loop and extracts a sequence interpolant at each iteration (as long as no counterexample is found), incrementally constructing a trace. The sequence interpolant represents an over-approximation of reachable states at an increasing number of steps.

ISB starts by checking the initial states do not violate the property. Assuming no counterexample of length 0 exists, ISB initializes the trace to $\vec{F} := \langle F_0 = Init \rangle$. It then operates just like BMC. In its first iteration, it solves φ^1. If the formula is satisfiable, a counterexample is found, and the algorithm terminates. Otherwise, a sequence interpolant is extracted for $A_1 = Init \wedge Tr$ and $A_2 = \neg P'$. In this case, the sequence contains the interpolant I_1^1. The interpolant represents an over-approximation of the states reachable from $Init$ after one transition ($Init \wedge Tr \Rightarrow I_1^1$), and in addition these states satisfy the property ($I_1^1 \wedge \neg P'$ is unsatisfiable). ISB then defines $F_1 = I_1^1[V^1 \leftarrow V]$, and the result is a safe trace $\vec{F} := \langle F_0, F_1 \rangle$.

Let us assume that ISB is in its kth iteration. At the kth iteration, the trace is $\vec{F} := \langle F_0, \ldots, F_{k-1} \rangle$ and φ^k is checked. The goal of the kth iteration is to extend the trace with a new element F_k. If φ^k is satisfiable, a counterexample is found, and the algorithm terminates. In case it is unsatisfiable, a sequence interpolant $\vec{I}^k := \langle I_1^k, \ldots, I_k^k \rangle$ is extracted with respect to $\vec{A}$ where $\vec{A}_1 = Init(V^0) \wedge Tr(V^0, V^1)$, $\vec{A}_i = Tr(V^{i-1}, V^i)$ for $2 \leq i \leq k$ and $\vec{A}_{k+1} = \neg P(V^k)$ (see Fig. 7). This sequence interpolant is used to extend the trace. The ith element of the existing trace $\vec{F}$ is updated by defining $F_i = F_i \wedge I_i^k[V^i \leftarrow V]$ for $1 \leq i < k$ and F_k to be I_k^k ($F_k = I_k^k[V^k \leftarrow V]$). The result is a safe trace of length k. It is left as an exercise for the reader to show why $\vec{F}$ is a safe trace.

At the end of the kth iteration, if an inductive invariant is found (Lemma 2), the algorithm terminates concluding that M is safe. Otherwise, the next iteration is executed.

```
1  Function ISB (M, P, N)
2      if ISSAT(φ^0) then
3          return CEX;
4      end
5      k ← 1;
6      F⃗ ← ⟨Init⟩;
7      while k <= N do
8          A⃗^k ← MKSEQ(φ^k);
9          if ISSAT(A⃗^k) then
10             return CEX;
11         end
12         I⃗^k ← SEQITP(A⃗^k);
13         i ← 1;
14         while i < k do
15             F⃗_i ← F⃗_i ∧ I⃗^k_i;
16             i ← i + 1;
17         end
18         F⃗.EXTEND(I⃗^k_k);
19         if F⃗.ISCLOSED() then
20             return SAFE;
21         end
22         k = k + 1;
23     end
24 end
25 return P holds up to N
```

Algorithm 3: ISB for a transition system M up to bound N

The pseudocode for ISB appears in Algorithm 3. The procedure MKSEQ partitions the BMC formula as described in the text. Moreover, the procedure ISSAT($\vec{A}^k$) operates on the conjunction of the elements of the trace. Note also that when $\vec{F}$ is closed, an inductive invariant is found (Definition 4 and Lemma 2). A detailed description of ISB appears in Vizel and Grumberg (2009) and Cabodi et al. (2011).

Interpolation-Based Model Checking (ITP)

ITP (McMillan 2003) is a complete SAT-based model checking algorithm that relies on interpolation to compute the trace. Similarly to ISB, ITP is based on BMC; however, the BMC queries used by ITP are different. Therefore, unlike ISB, it is not integrated directly into the BMC loop.

As it was shown, BMC formulates the question: "Does M have a counterexample of length k?" as a propositional formula φ^k (Formula 2). In a similar manner, BMC can also be formulated using the question "Does M have a counterexample of length i such that $1 \leq i \leq k$?" by using the following propositional formula:

Formula 4. $\psi^k \stackrel{\text{def}}{=} Init(V^0) \wedge path^{0,k} \wedge (\bigvee_{i=1}^{k} \neg P(V^i))$

In the original description of ITP (McMillan 2003), Formula 4 is used. ITP uses nested loops where the inner loop computes a *safe* trace by repeatedly checking formulas of the form ψ^k with a fixed k, and the outer loop increases the bound k when needed. The safe trace is computed inside the inner loop by extracting interpolants from unsatisfiable BMC formulas. Let us now describe the nested loops in more detail:

- **Inner Loop:** In general, the inner loop checks a fixed-bound BMC formula. At the first iteration, ψ^k is checked. If this BMC formula is satisfiable, then a counterexample exists, and the algorithm terminates. If it is unsatisfiable, then the following (A, B) pair is defined:
  - $A \stackrel{\text{def}}{=} Init(V^0) \wedge Tr(V^0, V^1)$
  - $B \stackrel{\text{def}}{=} path^{1,k} \wedge (\bigvee_{i=1}^{k} \neg P(V^i))$

  Following the interpolant definition, an interpolant I_1^k is extracted. The interpolant represents an over-approximation of the states reachable from $Init$ after one transition ($A \Rightarrow I_1^k$). In addition, no counterexample can be reached from I_1^k in $k-1$ transitions or less ($I_1^k \wedge B$ is unsatisfiable), which also guarantees that $I_1^k \Rightarrow P$. (Note that if, instead of ψ^k, φ^k would have been used, the interpolant would not necessarily satisfy P.) Thus, the sequence $\langle Init, I_1^k[V^1 \leftarrow V]\rangle$ is a valid safe trace. In the subsequent iterations, the formula $\psi^k[Init \leftarrow I_{j-1}^k]$ is checked, where j is the iteration of the inner loop. Thus, in the jth iteration, if $\psi^k[Init \leftarrow I_{j-1}^k]$ is unsatisfiable, an interpolant I_j^k is extracted with respect to the (A, B) pair where $A = I_{j-1}^k(V^1 \leftarrow V^0) \wedge Tr(V^0, V^1)$ and B is as before. Following this definition, I_j^k is an over-approximation of states reachable from I_{j-1}^k in one transition, and $\langle Init, I_1^k, \ldots, I_j^k\rangle$ is a safe trace. The inner loop terminates either when the BMC formula it checks is satisfiable or when an inductive invariant is found. In the latter case, the algorithm terminates concluding that the transition system is safe. In the former case, there are two cases to handle: if the BMC formula is satisfiable in the first iteration, a counterexample exists and the algorithm terminates; otherwise, the control is passed back to the outer loop, which increases k.
- **Outer Loop:** After the first iteration of the inner loop, over-approximated sets of reachable states are used as the initial condition of the checked BMC formulas. Thus, in case such a BMC formula becomes satisfiable, it is not clear if it is due to the existence of a counterexample or due to the over-approximation. When a BMC formula that uses an over-approximated set of states as the set of initial states becomes satisfiable, the control goes back to the outer loop that increases the bound k used for the BMC queries. Increasing k helps to either find a real counterexample or to increase the precision of the over-approximation.

  Note that B represents all bad states and all states that can reach a bad state in $k-1$ transitions or less. Therefore, when k is increased, the precision of the computed interpolant is also increased. For a sufficiently large k, the approximation obtained through interpolation becomes precise enough such that

the inner loop is guaranteed to find an inductive invariant if the system is safe (McMillan 2003), leading to the termination of the algorithm.

A detailed comparison between ITP and ISB appears in Vizel and Grumberg (2009).

Property Directed Reachability

The introduction of PDR (Bradley 2011; Een et al. 2011) has drastically changed the way that SAT-based model checking is perceived. Usually referred to as the "monolithic" approaches, interpolation-based techniques use SAT-solving as a blackbox that can either find a satisfying assignment or generate a proof of unsatisfiability. Intuitively, the proof of unsatisfiability represents a way to generalize a bounded proof into a candidate inductive invariant. While interpolation-based approaches utilize the full strength of state-of-the-art SAT solvers, there is little control over the performed generalization or the "inductiveness" of the generated candidate.

PDR, however, waives some of the strengths of the SAT solver and in return gains control over the generation of the candidate inductive invariant. This is achieved by employing a very specific search strategy. PDR's search strategy is based on a backward search that starts from the unsafe (or "bad") states in $\neg P$. The algorithm maintains a monotonic safe trace $\vec{F} := F_0, \ldots, F_k$, where each frame F_i over-approximates the set of states reachable from $Init$ in up to i steps of Tr. In addition, PDR maintains a queue Q of *proof obligations* $\langle s, i \rangle$, where s is a state that can reach a bad state in some number of transitions and i is the *level* of s, defined as the index of the smallest frame of the trace containing s (as the trace is monotonic, this means that $s \in F_i \setminus F_{i-1}$). At each iteration, PDR picks a proof obligation $\langle s, i \rangle$ from Q, prioritizing proof obligations with lower level. Then PDR tries to find a one-step predecessor of s in F_{i-1} and add it to Q with level $i - 1$. If at any point Q contains an initial state, then by construction there is a counterexample path from an initial state to a bad state. If no one-step predecessor of s exists, then PDR "blocks" s using a process called *inductive generalization*. The generalization technique yields a clause that is inductive relative to F_{i-1}, which is then used to strengthen the frames $F_0, \ldots, F_i$ of the trace, excluding s from these over-approximations (see Fig. 8). The algorithm terminates if either a counterexample is found or a frame is determined to be an inductive invariant that proves the property.

Notably, the SAT queries made by PDR involve only a single step of the transition relation. Each state s is represented by a conjunction of literals over V whose only satisfying assignment corresponds to s; accordingly, its negation $\neg s$ is a clause. Consequently, the SAT queries performed by PDR are computationally cheap (in comparison to ITP).

In the following, PDR is described in more detail. The PDR algorithm (Bradley 2011) iteratively refines and extends a monotonic safe trace where the frames are in CNF. In each (outer) iteration, the algorithm performs one of two actions:

- If no unsafe state is reachable from F_k in a single step (i.e., $F_k \wedge Tr \Rightarrow P'$), the algorithm extends the trace $\vec{F}$ with an additional frame $F_{k+1} = P$. F_{k+1} becomes the new *frontier*, and the resulting trace remains a monotonic safe trace. As an additional optimization, for each frame F_i, $0 \leq i \leq k$, PDR "pushes" clauses from F_i to F_{i+1}: if $c \in F_i$ and $F_i \wedge Tr \Rightarrow c'$, then c is added to F_{i+1} as well (where c' is obtained by replacing all variables in c with their primed counterparts). PDR ensures that the clauses in each frame F_{i+1} (for $0 \leq i < k$) are a subset of the clauses of its predecessor F_i, enabling efficient syntactic checks for the equality of frames. If during the propagation it is discovered that $F_{i+1} = F_i$ for some i, the algorithm terminates since F_i is an inductive invariant proving the safety of the transition system.
- If $F_k \wedge Tr \not\Rightarrow P'$, an unsafe state is reachable from F_k in a single step. A predecessor s in F_k of an unsafe state can be extracted from a satisfying assignment of $F_k \wedge Tr \wedge \neg P'$. The state s constitutes a *counterexample to induction* (CTI), since it demonstrates that F_k is not inductive. Note that F_{k-1} does not contain s, since its unsafe successor state would otherwise be reachable from F_{k-1}, violating the properties of a monotonic safe trace. Subsequently, PDR will try to prove the unreachability of s from F_{k-1} and adds a new proof obligation $\langle s, k \rangle$ to the priority queue Q used to store proof obligations.

  The algorithm now iteratively chooses a proof obligation $\langle s, i \rangle$ with the lowest level and checks whether the clause $\neg s$ is inductive relative to F_{i-1} by means of the *consecution query* $F_{i-1} \wedge \neg s \wedge Tr \Rightarrow \neg s'$. If this attempt fails, a CTI t in F_i (a predecessor of s, as illustrated in Fig. 8) can be extracted from a satisfying assignment of $F_{i-1} \wedge Tr \wedge s'$. The algorithm maintains that $F_0 = Init$, so a predecessor of a proof obligation with $i = 1$ represents a violation of the property P. Otherwise ($i > 1$), PDR will try to prove the unreachability of t from F_{i-2} and will add new proof obligations $\langle t, i-1 \rangle$ to Q. For efficiency, it is important to generalize t to a larger set of states, all of which can still reach s in a single step. This generalization is usually performed either using ternary simulation (Een et al. 2011) or an additional *lifting* SAT query (Chockler et al. 2011).

  On the other hand, if the consecution query holds, that is, if $\neg s$ is inductive relative to F_{i-1}, then PDR strengthens the frames $F_1, \ldots, F_i$ by excluding s and removes the proof obligation $\langle s, i \rangle$ from Q. To accelerate convergence, PDR deploys a *generalization* algorithm (Hassan et al. 2013) to find a clause $c \subseteq \neg s$ such that $F_{i-1} \wedge c \wedge Tr \Rightarrow c'$, which is added to all frames F_j, $1 \leq j \leq i$.

Example 5. Back to our running example, suppose that the property being verified is $P : c \leq 10$. Let us assume that just before its fourth (outer) iteration, PDR has constructed the trace $\vec{F} := \langle F_0, F_1, F_2, F_3 \rangle$, with $F_0 = Init = \{c = 0\}$, $F_1 = \{c \leq 1\}$, $F_2 = \{c \leq 2\}$, and $F_3 = \{c \leq 3\}$. One can easily verify that $\vec{F}$ is indeed a safe inductive trace.

PDR starts a new iteration and checks whether $F_3 \wedge Tr \Rightarrow P'$ holds:

$$(c \leq 3) \wedge (c' = (c == 8)\ ?\ 0 : c + 1) \wedge (c' > 10).$$

Clearly, this formula is unsatisfiable. PDR adds a new frame $F_4 = \{c \leq 10\}$ to the trace, resulting in $\vec{F} := \langle F_0, F_1, F_2, F_3, F_4 \rangle$. The pushing optimization is applied, but no clause can be propagated forward.

PDR starts another iteration and checks whether $F_4 \wedge Tr \Rightarrow P'$ holds:

$$(c \leq 10) \wedge (c' = (c == 8)\ ?\ 0 : c + 1) \wedge (c' > 10).$$

This formula is clearly satisfiable by assigning $c = 10$ and $c' = 11$. A new proof obligation $\langle c = 10, 4 \rangle$ is created and added to the priority queue Q. PDR now tries to find a predecessor to $c = 10$ in F_3 by checking whether $F_3 \wedge Tr \wedge c' = 10$ is satisfiable:

$$(c \leq 3) \wedge (c' = (c == 8)\ ?\ 0 : c + 1) \wedge (c' = 10).$$

This formula is unsatisfiable, and hence this proof obligation is blocked by adding $c \neq 10$ to all frames F_1, F_2, F_3, F_4 (though semantically only F_4 is changed). After this update, $F_4 = \{c \leq 10, c \neq 10\}$. PDR now goes back to checking if $F_4 \wedge Tr \Rightarrow P'$ holds:

$$(c \leq 10) \wedge (c \neq 10) \wedge (c' = (c == 8)\ ?\ 0 : c + 1) \wedge (c' > 10).$$

As this formula is unsatisfiable, PDR adds a new frame $F_5 = \{c \leq 10\}$ to the trace $\vec{F}$. The pushing optimization is applied but does not succeed in pushing the clause $c \neq 10$ from F_4 to F_5, as

$$(c < 10) \wedge (c' = (c == 8)\ ?\ 0 : c + 1) \wedge (c' = 10)$$

is satisfiable (with $c = 9$). At this point, the trace is $\vec{F} := \langle F_0, F_1, F_2, F_3, F_4, F_5 \rangle$, with $F_0 = Init = \{c = 0\}$, $F_1 = \{c \leq 1\}$, $F_2 = \{c \leq 2\}$, $F_3 = \{c \leq 3\}$, $F_4 = \{c \leq 10, c \neq 10\}$, and $F_5 = \{c \leq 10\}$.

Similarly to the previous iteration, PDR now checks if $F_5 \wedge Tr \Rightarrow P'$ holds. It does not, and based on the satisfying assignment, a proof obligation $\langle c = 10, 5 \rangle$ is added to Q. PDR tries to find a predecessor for $c = 10$ in F_4 by checking if $F_4 \wedge Tr \wedge (c' = 10)$ is satisfiable:

$$(c < 10) \wedge (c' = (c == 8)\ ?\ 0 : c + 1) \wedge (c' = 10).$$

It is, and based on the satisfying assignment, PDR creates another proof obligation $\langle c = 9, 4 \rangle$. Since there is no predecessor for $c = 9$ in F_3, PDR blocks this proof obligation by adding $c \neq 9$ to F_4 (and the lower frames). The proof obligation $\langle c = 10, 5 \rangle$ is now also blocked as it no longer has a predecessor in F_4. Thus, PDR adds $c \neq 10$ to F_5. It is easy to see that $F_5 \wedge Tr \Rightarrow P'$ now holds. At this point, $\vec{F} := \langle F_0, F_1, F_2, F_3, F_4, F_5 \rangle$, with $F_0 = Init = \{c = 0\}$, $F_1 = \{c \leq 1\}$, $F_2 = \{c \leq 2\}$, $F_3 = \{c \leq 3\}$, $F_4 = \{c \leq 10, c \neq 10, c \neq 9\}$, and $F_5 = \{c \leq 10, c \neq 10\}$.

Now, pushing is applied, and PDR tries to push $c \neq 9$ from F_4 to F_5, that is, to check whether $F_4 \wedge Tr \Rightarrow (c' \neq 9)$ holds:

$$(c \leq 10) \wedge (c \neq 10) \wedge (c \neq 9) \wedge (c' = (c == 8)\ ?\ 0 : c + 1) \wedge (c' = 9).$$

This formula is unsatisfiable (recall that design has no transition from $c = 8$ to $c = 9$). Hence, after adding $c \neq 9$ to F_5, the frames F_4 and F_5 become equal, and an inductive invariant $(c \leq 10) \wedge (c \neq 10) \wedge (c \neq 9)$ is found.

The above example demonstrates how PDR cleverly directs the generation of the inductive invariant and how it replaces a few "monolithic" SAT queries used by interpolation-based techniques by a large number of "incremental" single-step queries. PDR owes additional performance improvements to numerous optimizations (Hassan et al. 2013; Gurfinkel and Ivrii 2015; Cabodi et al. 2017; Froleyks and Biere 2021) (to name a few).

Combining Interpolation and PDR

Interpolation-based model checking and PDR, described in the previous sections, are two of the most successful strategies for SAT-based model checking. At a high level, both strategies attempt to prove a property by finding a safe inductive invariant, based on constructing a safe inductive trace. However, the two strategies employ very different generalization techniques: interpolation relies on a proof of unsatisfiability of a BMC instance, while PDR uses single-step reachability queries. Both approaches have their own strengths and weaknesses.

The interpolation-based approach does not pose restrictions on the SAT solver's search strategy, thus leveraging advances in SAT and in interpolation. However, the technique does not offer much control over generalization. It is at the mercy of the choices made by the SAT solver (which provides a particular resolution proof) and of the procedure used to generate an interpolant. Furthermore, in practice interpolants tend to be large, posing additional limitations on their use.

On the other hand, PDR directly manages both the search for a counterexample and the construction of a safe inductive invariant. Conceptually, PDR is based on a backward search and can be seen as a SAT solver with a specific search strategy that is based on the BMC structure of the problem (Bayless et al. 2013). Execution traces are extended step by step, and *inductive generalization* is used to block states that cannot be extended further. While PDR offers many advantages, including incremental solving and fine-grained control over generalization, it is limited to a *local* search strategy that can be inefficient.

To remedy the weaknesses of each approach, some techniques (Wu et al. 2013; Vizel et al. 2013, 2015; Vizel and Gurfinkel 2014; Li et al. 2017; Krishnan et al. 2019) combine ideas from interpolation, PDR, and k-induction. The reader is referred to these papers for a more detailed description.

Summary

Safety model checking is the backbone of many formal verification tools. This section reviewed some of the most successful safety model checking algorithms. Our exposition closely follows the history of advancements in relative scalability: from explicit-state model checking to symbolic model checking with BDDs, to bounded model checking and k-induction, to interpolation-based approaches, and finally to PDR. It is important to note that in practice there is no "one algorithm to rule them all," and usually multiple algorithms are run in parallel on the same verification task. When one algorithm reaches a definitive answer, all other algorithms are terminated.

Algorithms for Liveness Properties

Introduction

Intuitively, a liveness property says that "a good thing must keep happening." In linear time temporal logic, liveness properties are commonly expressed in the form GFp, $(Gu \rightarrow v)$, or FGq. The property of the form GFp states that on every path the signal p must hold infinitely often, the property of the form $(Gu \rightarrow v)$ states that on every path every occurrence of u must be eventually followed by an occurrence of v, and the property of the form FGq states that on every path the signal q must eventually hold forever. A counterexample to a liveness property is necessarily of *infinite length*. Liveness verification often involves the use of *fairness constraints* $f_1, \ldots, f_m$, which restrict the set of valid counterexamples to infinite traces for which $f_1, \ldots, f_m$ hold infinitely often. Equivalently, a liveness property of the form GFp together with fairness constraints $f_1, \ldots, f_m$ can be written as $GFf_1 \wedge \ldots GFf_m \Rightarrow GFp$. General liveness properties, together with fairness constraints, may be reduced to either of the forms GFp or FGq using additional logic (Wolper et al. 1983). While various liveness checking algorithms may benefit from considering fairness constraints directly, in the present discussion of model checking algorithms, it is assumed that the liveness property is either of the form GFp or FGq, choosing the form that is best suited to the algorithm at hand.

Given a liveness property of the form GFp, a counterexample to such a property is an infinite trace on which p holds only finitely many times. Since the state space of hardware models is finite, such a counterexample can be represented by a lasso-shaped trace, consisting of a prefix from an initial state to a $\neg p$-state s and a repeating loop suffix from s back to itself, with $\neg p$ holding on every state of the loop suffix, as is illustrated in Fig. 9. Similarly, a counterexample to FGq is an infinite trace on which $\neg q$ occurs infinitely many times and can be represented by a lasso-shaped trace, consisting of a prefix from an initial state to a $\neg q$-state s and a repeating loop suffix from s back to itself (note that $\neg q$ needs to hold only once on the loop, and without loss of generality it is assumed that this happens at s).

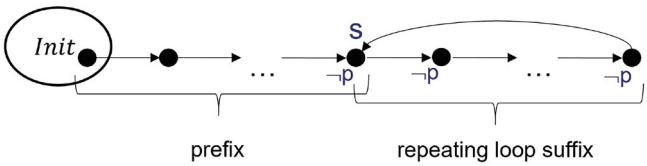


Fig. 9 A lasso-shaped counterexample to the liveness property GFp consisting of a prefix and a repeating loop suffix. An infinite-length counterexample to FGp can be obtained by (infinitely) unrolling the loop suffix

Table 2 Summary of liveness model checking algorithms

Algorithms	Details	Summary	Refs
SMC	Section "Symbolic Model Checking with BDDs"	Symbolic model checking using BDDs. Efficient on small designs. Does not scale to large designs	Burch et al. (1990); Ravi et al. (2000)
L2S	Section "Liveness-to-Safety Conversion (L2S)"	Liveness-to-safety conversion (reducing liveness verification to safety verification). Extremely useful in practice. However, the obtained safety properties may be hard to verify	Schuppan and Biere (2004)
kLiveness	Section "Bounded Liveness Checking"	Iteratively constructs a sequence of "bounded liveness" safety properties. Quite efficient in practice	Claessen and Sörensson (2012)
FAIR	Section "FAIR"	Incrementally learns information about reachable states and strongly connected regions of the state space. Quite efficient in practice	Bradley et al. (2011)

Even though liveness checking and safety checking are in the same complexity class, liveness checking is known to be significantly less scalable in practice.

Overview of Model Checking Algorithms

Table 2 summarizes some of the most widely used techniques for liveness verification. The following sections provide additional details.

Symbolic Model Checking with BDDs

Similar to safety model checking with BDDs described in section "Symbolic Model Checking (with BDDs)", liveness properties can be solved with BDDs via suitable fixed-point computations. The reader is referred to Ravi et al. (2000) for the detailed description and comparison of various BDD-based liveness algorithms.

Non-surprisingly, BDD-based algorithms are efficient only for designs with up to several hundred state elements.

Liveness-to-Safety Conversion (L2S)

The liveness-to-safety conversion (Schuppan and Biere 2004) is a general technique that converts a liveness problem to an equisatisfiable safety problem. Consider a liveness property of the form GFp. The technique is schematically presented in Fig. 10. It uses a "saved" state element for every original state element and nondeterministically selects when to "save" the original state. This "saved" state may be compared to later states, allowing to detect state repetitions. The new safety property asserts that if a state repetition was observed, then the signal p occurred at least once during the loop. Let's denote this translation as L2S.

To validate correctness of L2S, note that a counterexample to the safety property is a finite path with a state repetition $s_0 = s_1$ and with p not occurring from s_0 to s_1. It can be also seen as a lasso-shaped path, with a loop from s_0 back to itself, and without p occurring on the loop, and hence also represents a counterexample to GFp. Conversely, if the safety property holds, then on every potential loop p must occur at least once, showing that GFp holds as well.

Example 6. Consider the property $GF(c == 8)$ in our running example (this is property $P3$). The liveness-to-safety conversion produces the safety property that checks whether there is an execution with a state repetition but without c being 8 anywhere on the loop. Such an example can be easily found, for instance, $0000 \rightarrow 0001 \rightarrow 0000$. This means that $GF(c == 8)$ does not hold. Furthermore, by infinitely unrolling the loop from 0000 back to itself, one obtains an infinite counterexample to $GF(c == 8)$.

A crucial advantage of L2S is that by converting a liveness property to a safety property, any safety model checking algorithm can be used to solve the problem (such as the algorithms presented in section "Algorithms for Safety Properties"). In addition, any progress to safety checking immediately transfers to liveness checking as well. In practice, L2S is currently the most used liveness verification technique, and L2S followed by BMC is currently the best practical method to find a counterexample to a liveness property. Nevertheless, an important drawback of L2S

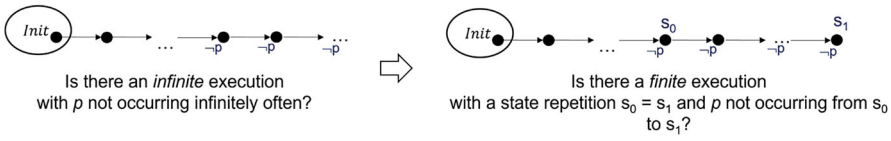


Fig. 10 Liveness-to-safety conversion: (left) the original liveness property, (right) the equivalent safety property

is that it doubles the number of state variables, substantially increasing the problem size and in practice yielding problems that are very hard to verify. Therefore, developing algorithms that can natively solve the original liveness problem remains an important research topic. Such algorithms will be presented in the following sections.

L2S with Abstraction The work Baumgartner and Mony (2009) describes an extension of L2S that trades completeness for smaller design size. It uses abstraction to choose a subset of state elements duplicated for the state repetition check. If the abstracted translation yields a proof, this constitutes a valid proof for the original translation (with all of the state elements duplicated). However, a counterexample for the abstracted translation may not represent a valid counterexample for the original translation, in which case refinement needs to be performed.

Bounded Liveness Checking

Counter-Based Translation

Recall that to prove that a liveness property GFp holds, one needs to prove that for every trace, starting from any state of the trace, the signal p must eventually occur. The *counter-based translation* in Schuppan and Biere (2004) is based on attempting to prove a stronger property $P_k = GF^k p$ for some value of k. The property P_k states that for every trace, starting from any state of the trace, the signal p must hold *within* k *steps* (or, equivalently, $\neg p$ cannot occur for k steps in a row). Each such property P_k is a safety property that can be solved by any safety model checker. Encoding P_k only requires k additional state elements (i.e., does not require doubling the number of state elements as in L2S). Given k, the proof of P_k yields the proof of the original property, but a counterexample to P_k may not represent a counterexample to the original property. Similarly to bounded model checking, a practical implementation of this idea is to start with the value of k being 0 and to increase the value of k by 1 each time that P_k is shown to be false. The technique was further improved in Claessen and Sörensson (2012) as discussed next.

kLiveness

For describing this and the following algorithms, it will now be more convenient to consider liveness properties of the form FGq. Recall that such a property asserts that on every trace the signal q must eventually hold forever, while an infinite-length counterexample to FGq is an execution for which $\neg q$ occurs infinitely many times.

The KLIVENESS algorithm from Claessen and Sörensson (2012) counts the maximal number of times that $\neg q$ can occur. Effectively, this technique checks a sequence of safety properties q_k which evaluate to false when q evaluates to false at least $k + 1$ times. Initially $q_0 = q$, and q_{k+1} is obtained from q_k by adding "absorbing logic" that masks one occurrence of $\neg q$. If for some value of k, the safety property q_k is proven valid, then on every (infinite) path the signal q can occur at

most k times, and thus the liveness property FGq is valid. Figure 11 illustrates the approach when $k = 2$.

A finite-length counterexample to q_k for some k does not guarantee the existence of a counterexample to the original liveness property or counterexamples for higher values of k. Though, if a finite-length counterexample happens to exhibit a state repetition within which q evaluates to false, then it is a valid counterexample to the original liveness property. Since the state space is finite, for suitably large k, either q_k will be proven or will yield a valid unbounded counterexample. KLIVENESS is thus sound and complete. As noted in Aleksandrowicz et al. (2013), in practice unbounded counterexamples can often be detected even for small values of k. Given the close relation between models being checked for increasing k, an *incremental* model checker such as IC3 offers the advantage of reusing information such as bounded and absolute invariants between each query.

Stabilizing Constraints A further contribution of Claessen and Sörensson (2012) is the generation of additional fairness constraints (called *stabilizing constraints*) that must hold on the loop of any potential counterexample to FGq. Each such stabilizing constraint is of the form FGr for some signal r. Given that FGr holds, it is possible to replace the original liveness property FGq by the property $FG(r \rightarrow q) \equiv FG(\neg r \vee q)$. Intuitively, it is harder for $(r \rightarrow q)$ to evaluate to false than for q. In practice, the additional stabilizing constraints discovered by KLIVENESS significantly increase the chances that the resulting safety query is valid for a small value of k.

Example 7. Consider the finite-state machine presented in Fig. 12. The path $000 \rightarrow 001$ is a path from an initial state on which $\neg q$ occurs once; hence, the safety

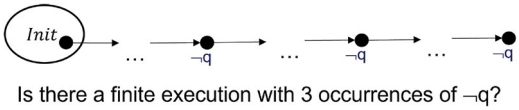


Fig. 11 A safety query produced by KLIVENESS for $k = 2$. If there is no execution for which $\neg q$ occurs three times, then FGq holds

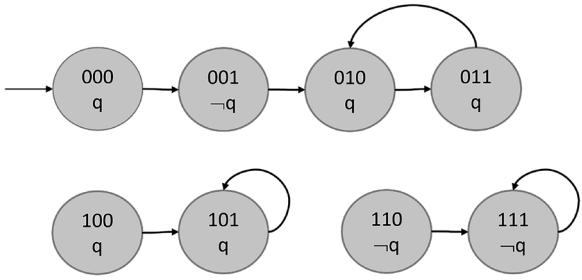


Fig. 12 A finite-state machine with 8 states. The signal q is true in states 000, 010, 011, 100, and 101 and false in states 001, 110, and 111

property q_0 (produced by kLIVENESS) does not hold. However, there is no path from an initial state on which $\neg q$ occurs two times; hence, q_1 holds, and FGq holds as well.

Furthermore, suppose there is some way to deduce that every possible counterexample to FGq must eventually always transition from the state 111 to itself (this is certainly the case here, as this is the only loop containing an occurrence of $\neg q$). Then $FG(s = 111)$ is a stabilizing constraint. As there is no path from an initial state on which $\neg((s = 111) \rightarrow q) \equiv (s = 111) \wedge q$ occurs even once, the kLIVENESS safety query holds even for $k = 0$.

FAIR

FAIR (Bradley et al. 2011) is an iterative algorithm that incrementally learns information about reachable states and the strongly connected regions of the state space. Roughly speaking, a *reachability assertion* R indicates that all the states on a potential lasso-shaped counterexample belong to R, while a *wall* W states that all states on the loop suffix of a potential counterexample either together belong to W or together belong to the complement of W. If one side of the wall W has no reachable states, the wall actually represents a constraint on all states on the loop of a potential counterexample or in other words is a *stabilizing constraint* as discussed earlier. Specializing to liveness properties of form FGq, FAIR uses a SAT solver to obtain a $\neg q$-state s, subject to the previously discovered reachability assertions and walls. If this query is unsatisfiable, then FGq holds. Otherwise, FAIR tries to compute a lasso-shaped counterexample for s, checking whether s is reachable from an initial state and then whether s can eventually transition to itself. If both queries are satisfiable, the liveness property fails. Otherwise, FAIR requires the underlying safety model checker to produce an inductive proof of unsatisfiability. If s is not reachable from an initial state, this proof represents a new reachability assertion. If s cannot transition to itself, this proof represents a new wall. In either case, the algorithm makes progress and must eventually terminate with a conclusive verification result. The core of the approach is illustrated in Fig. 13.

Example 8. Consider again the finite-state machine in Fig. 12. FAIR may discover that the states 110 and 111 cannot be reached from an initial state; this information represents a "reachability assertions." Additionally, FAIR may discover that the state 001 does not have a loop back to itself and thus may conclude that 001 cannot be on

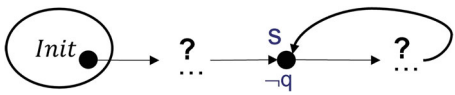


Fig. 13 Given a state s, FAIR creates two safety queries: one query checks whether there is a path from $Init$ to s, and one query checks whether there is a path from s back to itself

the loop suffix of any potential counterexample; this type of information represents a "wall." As all of the $\neg q$-states are now eliminated, FAIR will conclude that FGq holds.

Combining KLIVENESS and FAIR The two algorithms KLIVENESS and FAIR have different strengths. When FGq is valid, KLIVENESS works well when a small value of k is sufficient to prove unsatisfiability; otherwise, the underlying safety queries become unscalable as k becomes large. FAIR works well when inductive proofs restrict large portions of the search space; otherwise, too many iterations are required. The work Ivrii et al. (2018) shows how one may combine the strengths of both approaches.

Summary

Modern model checkers use a very diverse set of algorithms for solving liveness properties. These include BDD-based approaches, converting liveness properties to safety properties, constructing stronger bounded liveness properties, and exploring the structure of how different states in the design may reach each other. There is no best method overall, and a common practice is run a portfolio of different algorithms in parallel.

Design Simplification Techniques

Modern hardware model checkers support a large number of design transformation techniques. These techniques do not aim to solve the problem (though, in cases, they may) but rather to simplify it, by reducing the number of state elements, inputs, or auxiliary logic gates used to represent the problem. A typical model checker (for instance, ABC (Brayton and Mishchenko 2010)) is able to apply different transformation techniques in a sequence, each gradually chipping away at the problem size, until the problem can be handled by one of the safety or liveness model checking algorithms. Many useful transformations have been developed over the decades, ranging from area-reduction techniques from logic synthesis to abstractions and temporal transformations that apply only in a verification context. While a smaller problem size does not necessarily provide a guarantee that the problem is easier to solve, in practice simplified designs do tend to be simpler. The role of logic transformations to boost hardware verification is well established (e.g., Kuehlmann and Baumgartner 2001; Mony et al. 2004; Brayton and Mishchenko 2010), with the strength of the approach based upon the availability of a variety of different complementary transformations.

The design simplification techniques can be roughly classified into two types of algorithms: reductions and over-approximation techniques. Reduction algorithms (section "Reductions") attempt to reduce the problem to an equisatisfiable but smaller problem. Thus, a verification result for the reduced problem also applies

for the original problem. Over-approximation algorithms (section "Over-approximations") are based on removing details about the system that seem to be irrelevant to the checked property. By that, the abstract system has a smaller state space but may include more states than are reachable and hence include more behaviors. If a property holds for the abstracted system, it also holds for the original system. However, a failure for the abstracted system may not represent a real failure, and the result is inconclusive.

Reductions

This section gives a brief glimpse into the wealth of available reduction techniques. Most model checkers internally represent the problem as a sequential circuit, with the *And-Inverter Graph* format (Kuehlmann et al. 2002) being especially popular. A circuit in this format contains only constants, primary inputs, two-input AND-gates, inverters, and registers. Thus, the reduction techniques explicitly aim to simplify this circuit, by minimizing the number of registers, inputs, and AND-gates while guaranteeing that the simplified problem is equisatisfiable to the original one. Reducing the number of registers is especially important, as this automatically reduces the search space which the algorithms like PDR need to explore.

Combinational Redundancy Removal

The combinational redundancy removal techniques eliminate redundant gates and perform logic rewriting to replace logic cones with smaller functionally equivalent or observably equivalent cones. The *cone-of-influence* reduction removes logic that is completely irrelevant to the property. *Constant propagation* simplifies the design by propagating constant values. *Structural hashing* is used to merge structurally equivalent AND-gates. *AIG balancing, rewriting, and refactoring* apply DAG-aware transformations to reduce the AIG size (Bjesse and Borälv 2004; Mishchenko et al. 2006). Resource-bounded *BDD- and SAT-based analysis* is used to detect functionally redundant gates (Kuehlmann et al. 2002).

Retiming

Retiming attempts to reduce the number of registers by relocating them across combinational gates (Hurst et al. 2007; Kuehlmann and Baumgartner 2001; Baumgartner and Kuehlmann 2001), in such a way that the number of registers does not change on each path from a primary input to a primary output and on each loop.

Sequential Redundancy Removal

The *correspondence-checking* algorithms try to detect and merge functionally equivalent gates (van Eijk 1998; Mony et al. 2009; Baumgartner et al. 2006). This is a very powerful reduction technique in general and is especially useful for *combinational and sequential equivalence checking*, where two similar designs are being compared. For efficiency, the approach is based on first guessing redundancy

candidates, for example, using symbolic simulation, and then on using induction to prove these redundancies. Already proven redundancies can be exploited to prove additional redundancies.

Input Reparameterization

Range-preserving parametric-reencoding procedures replace logic cones adjacent to input variables with behaviorally equivalent logic cones with fewer input variables (Baumgartner 2002; Moon et al. 2002; Eén and Mishchenko 2013). One specific approach (Baumgartner 2002) is based on identifying a min-cut between the inputs and the sequential elements and targets and replacing that cut with a simpler resynthesized logic cone which comprises at most as many input variables as the cut width. A faster but lossier approach (Eén and Mishchenko 2013) checks if some inputs can be set to constant values while preserving the behavior of the combinational part of the circuit. This transformation often reduces the gate count as well.

Phase Abstraction

Phase abstraction performs a structural state folding and clocking abstraction used to eliminate the verification overhead of "clocked" designs, where each clock period comprises multiple verification time steps modeled using an oscillating clock (Baumgartner 2002; Bjesse and Kukula 2005). At a high level, the algorithm searches for oscillating clock-like logic in the circuit and eliminates this logic by unfolding next-state functions of the sequential elements modulo their periodicity. When applicable, this often eliminates many registers and often greatly enhances the potential of other reduction techniques.

Over-approximations

Abstraction (Clarke et al. 1992) is a widely used method to mitigate the state explosion problem. It aims to reduce the state space by removing details about the system that seem irrelevant to the checked property, that is, do not seem to be necessary either for verification or for refutation. Abstraction is best suited for proving properties. For valid properties, in practice up to 90% of design's state elements can be safely removed (preserving validity of the result), making the abstraction substantially more powerful than the reduction techniques considered previously. However, abstraction is ultimately an over-approximation technique that introduces additional behaviors not present in the original design. Determining which details about the system can be ignored without introducing erroneous (also called *spurious*) behaviors is not an easy task.

The most common abstraction techniques described below are based on the intuition that for a suitable bound k, the logic sufficient to prove the property for the first k steps of the design is also sufficient for an unbounded proof.

Proof-Based Abstraction

Proof-Based Abstraction (PBA) (McMillan and Amla 2003) is a top-down approach that considers the concrete transition system M and constructs an abstract model after verifying that no counterexample exists up to a specific length. The approach uses the ability of SAT solvers to compute an *unsatisfiable core* of an unsatisfiable formula, that is, a subset of the formula's clauses sufficient for unsatisfiability (Goldberg and Novikov 2003; Eén et al. 2010).

PBA is based on the BMC loop (see section "Bounded Model Checking"). At each iteration, the formula φ^k (Formula 2) is checked using a SAT solver. If φ^k is satisfiable, then a counterexample is found. Otherwise, φ^k is unsatisfiable, and an unsatisfiable core $UC(\varphi^k)$ is extracted. Let us define the set $V_a = \{v \mid v^i \in \text{Vars}(UC(\varphi^k)), 0 \leq i \leq k\}$ as the set of variables from the transition system that appears in any of the unsatisfiable cores computed so far. Clearly, $V_a \subseteq V$. The abstract transition system M_a is derived from M by making all variables $v \in V \setminus V_a$ nondeterministic (i.e., leaving them unconstrained). This abstraction, in the above context, is usually referred to as a "visible variables" abstraction (Kurshan 1994). This abstract model can now be passed to a complete model checking algorithm for verification. If the property is proved on the abstract model M_a, then the property also holds on the original model M, and the algorithm terminates. However, a counterexample in M_a may not exist in M (due to the abstraction). In the case of a spurious counterexample, the validity of the property remains unknown, and PBA executes the next iteration of the BMC loop with a larger k.

Counterexample-Guided Abstraction

Counterexample-guided abstraction refinement (CEGAR) (Clarke et al. 2000, 2003) is a bottom-up approach. CEGAR starts with a coarse abstract model M_a and passes it to a model checking algorithm of choice. If a spurious counterexample is found, M_a is refined. Refinement makes sure the spurious counterexample is removed from M_a. Note that unlike PBA, CEGAR-based approaches do not remove all counterexamples of a given length. This process continues until either a real counterexample is found or M_a is proved to be safe with respect to the checked property.

There are many variants of this framework. For example, CEGAR can be used in conjunction with BMC in a similar manner to how PBA works. In this case, CEGAR starts with a coarse abstract model M_a and the unfolding depth $k = 1$. During the BMC stage, it looks for length-k counterexamples and uses spurious counterexamples to refine M_a. When there are no counterexamples at a given unfolding depth k, the unfolding depth is increased. The approach continues until either a real counterexample is found or the abstracted model M_a is deemed sufficiently adequate for proof analysis. In the latter case, M_a is passed to a complete model checking algorithm for verification. As before, if M_a is safe with respect to the checked property, then so is M and the algorithm terminates, while in the case of a spurious counterexample the algorithm continues with a larger unfolding depth.

Other Approaches

It is possible to combine PBA and CEGAR into a single *hybrid* approach (Eén et al. 2010; Mishchenko et al. 2013). First, as per CEGAR, a sufficiently adequate abstract model is constructed bottom-up, refining spurious counterexamples up to a specific length k. Second, as per PBA, the abstraction is shrunk to the variables appearing in an unsatisfiable core and only then passed to a complete model checking algorithm. The advantage of this hybrid strategy is that it completely avoids performing BMC queries on the original model and hence may apply to designs with a very large number of variables.

In the description of PBA and CEGAR above, the abstraction was stated in terms of state variables. As the design is usually represented as a circuit, a more refined approach is possible, based on stating the abstraction in terms of internal gates (see Mishchenko et al. (2013)).

Summary

Modern model checkers use many different techniques to simplify and to abstract a design, prior to running a complete model checking algorithm. In practice, these techniques significantly improve verification run-times and are indispensable for large designs.

Conclusion

Ensuring that a design conforms to its specification is an indispensable part of the modern design automation flow. Bit-level model checking has gone a very long way since it was first introduced. Representing the system symbolically, employing complex forms of inductive reasoning, advances in SAT-solving, and simplifying and abstracting the design – all of this pushed the scalability of modern hardware verification tools to designs with thousands of state elements and hundreds of thousands of gates.

However, with the move to system on a chip (SoC), the size and complexity of modern designs continue to rapidly grow. This requires model checking tools to continue delivering performance boosts, and the research in this domain remains highly important. Historically, every decade or so, there is a big leap forward in performance for model checking. The authors eagerly await the upcoming advancements.

References

Aleksandrowicz G, Baumgartner J, Ivrii A, Nevo Z (2013) Generalized counterexamples to liveness properties. In: Formal methods in computer-aided design, FMCAD 2013, Portland, 20–23 Oct 2013. IEEE, pp 169–180

Baumgartner J (2002) Automatic structural abstraction techniques for enhanced verification. PhD thesis, University of Texas

Baumgartner J, Kuehlmann A (2001) Min-area retiming on dynamic circuit structures. In: Ernst R (ed) Proceedings of the 2001 IEEE/ACM international conference on computer-aided design, ICCAD 2001, San Jose, 4–8 Nov 2001. IEEE Computer Society, pp 176–182

Baumgartner J, Mony H (2009) Scalable liveness checking via property-preserving transformations. In: Benini L, Micheli GD, Al-Hashimi BM, Müller W (eds) Design, automation and test in Europe, DATE 2009, Nice, 20–24 Apr 2009. IEEE, pp 1680–1685

Baumgartner J, Mony H, Paruthi V, Kanzelman R, Janssen G (2006) Scalable sequential equivalence checking across arbitrary design transformations. In: 24th international conference on computer design (ICCD 2006), 1–4 Oct 2006, San Jose. IEEE, pp 259–266

Bayless S, Val CG, Ball T, Hoos HH, Hu AJ (2013) Efficient modular SAT solving for IC3. In: Formal methods in computer-aided design (FMCAD). IEEE, pp 149–156

Biere A, Cimatti A, Clarke EM, Zhu Y (1999) Symbolic model checking without BDDs. In: Tools and algorithms for the construction and analysis of systems (TACAS). LNCS, vol 1579. Springer, pp 193–207

Bjesse P, Borälv A (2004) Dag-aware circuit compression for formal verification. In: 2004 international conference on computer-aided design, ICCAD 2004, San Jose, 7–11 Nov 2004. IEEE Computer Society/ACM, pp 42–49

Bjesse P, Kukula JH (2005) Automatic generalized phase abstraction for formal verification. In: 2005 international conference on computer-aided design, ICCAD 2005, San Jose, 6–10 Nov 2005. IEEE Computer Society, pp 1076–1082

Bradley AR (2011) SAT-based model checking without unrolling. In: Verification, model checking and abstract interpretation (VMCAI). LNCS, vol 6538. Springer, pp 70–87

Bradley AR, Somenzi F, Hassan Z, Zhang Y (2011) An incremental approach to model checking progress properties. In: Bjesse P, Slobodová A (eds) International conference on formal methods in computer-aided design, FMCAD'11, Austin, 30 Oct–02 Nov 2011. FMCAD Inc., pp 144–153

Brayton RK, Mishchenko A (2010) ABC: an academic industrial-strength verification tool. In: Computer aided verification (CAV). LNCS, vol 6174. Springer, pp 24–40

Bryant RE (1986) Graph-based algorithms for Boolean function manipulation. IEEE Trans Comput 35(8):677–691

Burch JR, Clarke EM, McMillan KL, Dill DL, Hwang LJ (1990) Symbolic model checking: 10^{20} states and beyond. In: Logic in computer science (LICS). IEEE, pp 428–439

Cabodi G, Nocco S, Quer S (2011) Interpolation sequences revisited. In: Design automation and test in Europe (DATE). IEEE, pp 316–322

Cabodi G, Camurati P, Mishchenko A, Palena M, Pasini P (2017) SAT solver management strategies in IC3: an experimental approach. Formal Methods Syst Des 50(1):39–74

Chockler H, Ivrii A, Matsliah A, Moran S, Nevo Z (2011) Incremental formal verification of hardware. In: Formal methods in computer-aided design (FMCAD). FMCAD Inc., pp 135–143

Claessen K, Sörensson N (2012) A liveness checking algorithm that counts. In: Cabodi G, Singh S (eds) Formal methods in computer-aided design, FMCAD 2012, Cambridge, 22–25 Oct 2012. IEEE, pp 52–59

Clarke EM, Emerson EA, Sistla AP (1986) Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans Program Lang Syst 8(2):244–263

Clarke E, Grumberg O, Long D (1992) Model checking and abstraction. In: Principles of programming languages (POPL). ACM, pp 343–354

Clarke EM, Grumberg O, Jha S, Lu Y, Veith H (2000) Counterexample-guided abstraction refinement. In: Computer aided verification (CAV). LNCS, vol 1855. Springer, pp 154–169

Clarke EM, Grumberg O, Peled DA (2001) Model checking, 1st edn.. MIT Press

Clarke EM, Grumberg O, Jha S, Lu Y, Veith H (2003) Counterexample-guided abstraction refinement for symbolic model checking. J ACM 50(5):752–794

Clarke EM, Kroening D, Ouaknine J, Strichman O (2004) Completeness and complexity of bounded model checking. In: Verification, model checking and abstract interpretation (VMCAI). LNCS, vol 2937. Springer, pp 85–96

Cook SA (1971) The complexity of theorem-proving procedures. In: ACM symposium on theory of computing (STOC). ACM, pp 151–158

Craig W (1957) Linear reasoning. A new form of the Herbrand-Gentzen theorem. J Symb Logic 22(3):250–268

Eén N, Mishchenko A (2013) A fast reparameterization procedure. In: Ganai MK, Sen A (eds) Proceedings of the second international workshop on design and implementation of formal tools and systems, Portland, 19 Oct 2013. CEUR workshop proceedings, vol 1130. CEUR-WS.org

Eén N, Mishchenko A, Amla N (2010) A single-instance incremental SAT formulation of proof- and counterexample-based abstraction. In: Bloem R, Sharygina N (eds) Proceedings of 10th international conference on formal methods in computer-aided design, FMCAD 2010, Lugano, 20–23 Oct. IEEE, pp 181–188

Een N, Mishchenko A, Brayton R (2011) Efficient implementation of property directed reachability. In: Formal methods in computer-aided design (FMCAD). FMCAD Inc, pp 125–134

Froleyks N, Biere A (2021) Single clause assumption without activation literals to speed-up IC3. In: Formal methods in computer aided design, FMCAD 2021, New Haven, 19–22 Oct 2021. IEEE, pp 72–76

Goldberg E, Novikov Y (2003) Verification of proofs of unsatisfiability for CNF formulas. In: Design automation and test in Europe (DATE). IEEE, pp 886–891

Gurfinkel A, Ivrii A (2015) Pushing to the top. In: Kaivola R, Wahl T (eds) Formal methods in computer-aided design, FMCAD 2015, Austin, 27–30 Sept 2015. IEEE, pp 65–72

Hassan Z, Bradley AR, Somenzi F (2013) Better generalization in IC3. In: Formal methods in computer-aided design (FMCAD). FMCAD Inc., pp 157–164

Hurst AP, Mishchenko A, Brayton RK (2007) Fast minimum-register retiming via binary maximum-flow. In: Formal methods in computer-aided design, 7th international conference, FMCAD 2007, Austin, 11–14 Nov 2007, Proceedings. IEEE Computer Society, pp 181–187

Ivrii A, Nevo Z, Baumgartner J (2018) k-fair = k-liveness + FAIR revisiting sat-based liveness algorithms. In: Bjørner N, Gurfinkel A (eds) 2018 formal methods in computer aided design, FMCAD 2018, Austin, 30 Oct–2 Nov 2018. IEEE, pp 1–5

Jhala R, McMillan KL (2005) Interpolant-based transition relation approximation. In: Computer aided verification (CAV), vol 3576. Springer, pp 39–51

Krishnan HGV, Vizel Y, Ganesh V, Gurfinkel A (2019) Interpolating strong induction. In: Dillig I, Tasiran S (eds) Computer aided verification – 31st international conference, CAV 2019, New York City, 15–18 July 2019, Proceedings, Part II. Lecture notes in computer science, vol 11562. Springer, pp 367–385

Kuehlmann A, Baumgartner J (2001) Transformation-based verification using generalized retiming. In: Berry G, Comon H, Finkel A (eds) Computer aided verification, 13th international conference, CAV 2001, Paris, 18–22 July 2001, Proceedings. Lecture notes in computer science, vol 2102. Springer, pp 104–117

Kuehlmann A, Paruthi V, Krohm F, Ganai MK (2002) Robust boolean reasoning for equivalence checking and functional property verification. IEEE Trans Comput Aided Des Integr Circuits Syst 21(12):1377–1394

Kurshan RP (1994) Computer-aided verification of coordinating processes: the automata-theoretic approach. Princeton University Press, Princeton

Li J, Zhu S, Zhang Y, Pu G, Vardi MY (2017) Safety model checking with complementary approximations. In: Parameswaran S (ed) 2017 IEEE/ACM international conference on computer-aided design, ICCAD 2017, Irvine, 13–16 Nov 2017. IEEE, pp 95–100

McMillan KL (2003) Interpolation and SAT-based model checking. In: Computer aided verification (CAV). LNCS, vol 2725. springer, pp 1–13

McMillan KL, Amla N (2003) Automatic abstraction without counterexamples. In: Tools and algorithms for the construction and analysis of systems (TACAS). LNCS, vol 2619. Springer, pp 2–17

Mishchenko A, Chatterjee S, Brayton RK (2006) Dag-aware AIG rewriting a fresh look at combinational logic synthesis. In: Sentovich E (ed) Proceedings of the 43rd design automation conference, DAC 2006, San Francisco, 24–28 July 2006. ACM, pp 532–535

Mishchenko A, Eén N, Brayton RK, Baumgartner J, Mony H, Nalla PK (2013) GLA: gate-level abstraction revisited. In: Design automation and test in Europe (DATE). EDA Consortium, pp 1399–1404

Mony H, Baumgartner J, Paruthi V, Kanzelman R, Kuehlmann A (2004) Scalable automated verification via expert-system guided transformations. In: Hu AJ, Martin AK (eds) Formal methods in computer-aided design, 5th international conference, FMCAD 2004, Austin, 15–17 Nov 2004, Proceedings. Lecture notes in computer science, vol 3312. Springer, pp 159–173

Mony H, Baumgartner J, Mishchenko A, Brayton RK (2009) Speculative reduction-based scalable redundancy identification. In: Benini L, Micheli GD, Al-Hashimi BM, Müller W (eds) Design, automation and test in Europe, DATE 2009, Nice, 20–24 Apr 2009. IEEE, pp 1674–1679

Moon I, Kwak H, Kukula JH, Shiple TR, Pixley C (2002) Simplifying circuits for formal verification using parametric representation. In: Aagaard MD, O'Leary JW (eds) Formal methods in computer-aided design, 4th international conference, FMCAD 2002, Portland, 6–8 Nov 2002, Proceedings. Lecture notes in computer science, vol 2517. Springer, pp 52–69

Pnueli A (1977) The temporal logic of programs. In: 18th annual symposium on foundations of computer science, Providence, 31 Oct–1 Nov 1977. IEEE Computer Society, pp 46–57

Queille J-P, Sifakis J (1982) Specification and verification of concurrent systems in CESAR. In: International symposium on programming, pp 337–351

Ravi K, Bloem R, Somenzi F (2000) A comparative study of symbolic algorithms for the computation of fair cycles. In: WAH. Jr. and Johnson SD (eds) Formal methods in computer-aided design, third international conference, FMCAD 2000, Austin, 1–3 Nov 2000, Proceedings. Lecture notes in computer science, vol 1954. Springer, pp 143–160

Rozier KY (2011) Linear temporal logic symbolic model checking. Comput Sci Rev 5(2):163–203

Schuppan V, Biere A (2004) Efficient reduction of finite state model checking to reachability analysis. Int J Softw Tools Technol Transf 5(2–3):185–204

Sheeran M, Singh S, Stålmarck G (2000) Checking safety properties using induction and a SAT-solver. In: Formal methods in computer-aided design (FMCAD). LNCS, vol 1954. Springer, pp 108–125

Tseitin G (1983) On the complexity of proofs in propositional logics. In: Siekmann J, Wrightson G (eds) Automation of reasoning: classical papers in computational logic 1967–1970, vol 2. Springer. Originally published 1970

van Eijk CAJ (1998) Sequential equivalence checking without state space traversal. In: Dewilde PM, Rammig FJ, Musgrave G (eds) 1998 design, automation and test in Europe (DATE'98), 23–26 Feb 1998, Le Palais des Congrès de Paris, Paris. IEEE Computer Society, pp 618–623

Vardi MY (2007) Automata-theoretic model checking revisited. In: Cook B, Podelski A (eds) Verification, model checking, and abstract interpretation, 8th international conference, VMCAI 2007, Nice, 14–16 Jan 2007, Proceedings. Lecture notes in computer science, vol 4349. Springer, pp 137–150

Vizel Y, Grumberg O (2009) Interpolation-sequence based model checking. In: Formal methods in computer-aided design (FMCAD). IEEE, pp 1–8

Vizel Y, Gurfinkel A (2014) Interpolating property directed reachability. In: Computer aided verification (CAV). LNCS, vol 8559. Springer, pp 260–276

Vizel Y, Ryvchin V, Nadel A (2013) Efficient generation of small interpolants in CNF. In: Computer aided verification (CAV). LNCS, vol 8044. Springer, pp 330–346

Vizel Y, Gurfinkel A, Malik S (2015) Fast interpolating BMC. In: Kroening D, Pasareanu CS (eds) Computer aided verification – 27th international conference, CAV 2015, San Francisco, 18–24 July 2015, Proceedings, Part I. Lecture notes in computer science, vol 9206. Springer, pp 641–657

Wolper P, Vardi MY, Sistla AP (1983) Reasoning about infinite computation paths (extended abstract). In: 24th annual symposium on foundations of computer science, Tucson, 7–9 Nov 1983. IEEE Computer Society, pp 185–194

Wu C, Wu C, Lai C, Huang CR (2013) A counterexample-guided interpolant generation algorithm for sat-based model checking. In: The 50th annual design automation conference 2013, DAC'13, Austin, 29 May–07 June 2013. ACM, pp 118:1–118:6

High-Level Formal Equivalence

35

Theo Drane and M. V. Achutha Kiran Kumar

Contents

Types of Equivalence to Check 1244
Combinational Equivalence 1245
Sequential Equivalence 1246
Transaction-Based Equivalence 1247
Advanced Datapath Verification 1259
Managing Inconclusive Proofs 1259
Accuracy Challenges 1261
References 1268

Abstract

FEV can be broadly defined as comparing two models to determine whether they are equivalent. In general, it is customary to refer to the two models being compared as the SPEC, or specification model, and the IMP, or implementation model. Typically, the SPEC will be the more abstract model: it may be an RTL model, an unoptimized schematic netlist, a reference model, or a description in a high-level modeling language. The IMP will usually be an equally or more concrete model for comparison: a more refined model, a new or updated RTL, or an optimized schematic netlist implementation.

T. Drane (✉)
Intel Corporation, Folsom, CA, USA
e-mail: theo.drane@intel.com

M. V. A. Kiran Kumar (✉)
DEG, Intel Corporation, Bengaluru, India
e-mail: achutha.kirankumar.v.m@intel.com

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_36

Keywords

CvsRTL · Formal equivalence · FEV · Convergence

The need for FEV can be seen as a natural outgrowth of the increased level of design abstraction as the industry has matured over the past half century. During the initial days of chip design (before 1980), the designers used to draw the circuits by hand and work at the transistor level. In the following decade (1981–1989), the drawing of such circuits was improved with the aid of computer-aided design (CAD), and circuit design became much simpler, but was not yet optimal. In these early days, nobody thought about verifying their designs formally; the technology had not yet reached the level where such tools were possible for practical use.

The RTL mode of designing started around the early 1990s and is still the major mode of logic design as of this writing. With the advent of RTL, new tools were introduced to automatically synthesize the coded design into the functional schematic netlists. This allowed the generation of much more complex netlists than were possible with previous methods, which necessitated good RTL-vs-netlist FEV tools. In the twenty-first century, the C/C++/SystemC mode of coding the design has begun to emerge, where the design is defined in the high-level language, and a new generation of synthesis tools can convert the high-level language to RTL and even to the netlist. Hence formal equivalence checks have become an even more important requirement to ensure that designs are faithfully translated across the different abstraction levels. Due to these various motivations, FEV is one of the most mature FV techniques; it is now considered a standard requirement to ensure the design intent is maintained as designers refine their abstractions into real designs.

In this chapter various equivalence techniques will be explored, look at their applications in the real world, and show how this requires a different style of interaction with the tools and with the design. The primarily context for the discussion talks about the usages where at least one of the models is RTL. These are also the usage modes for which the most mature industry tools are available.

As design and complexity grows, it is not unusual to encounter formal equivalency proofs that yield inconclusive results. Approaches to overcoming such challenges as well as cases of bounded equivalence are covered at the end of the chapter.

Types of Equivalence to Check

Before dwelling further on a detailed discussion of FEV techniques, it is important to define “equivalence.” There are several major notions of equivalence that are commonly used by current EDA tools: combinational equivalence, sequential equivalence, and transactional equivalence.

Combinational Equivalence

Combinational equivalence is the most mature FEV technique in the EDA industry; using this technique to compare RTL and schematic netlists is considered a requirement at most companies doing VLSI design. This is the primary FEV technique used for state-matching models, where every state element (latch or flop) in the SPEC corresponds to a specific state element in the IMP. In this mode, two designs are claimed to be equivalent when all combinational logic between any pair of states are logically equivalent. In other words, there is a 1:1 correspondence between the state elements of the two models. Whenever the equivalence of a pair of state elements in the two models is checked, the tool uses the assumption that corresponding state elements in their fanin cones will contain identical values at all points in time. In effect, every latch or flop in the two designs is being treated as a cut point, with verification only encompassing the logic between the state elements.

This might seem like a huge limitation – but it arose in tandem with logic synthesis technology that has a similar structure. In other words, most EDA tools that synthesize netlists from RTL are also state-matching, guaranteeing (mostly) that each state element in the netlist will correspond to a state element in the RTL. In addition, the state-matching compromise makes the FEV problem much more tractable: FEV tools that use this method just need to analyze Boolean expressions, with no need to account for value changes over time. Thus, combinational FEV is the technique used in most RTL to gate logic checkers today. If state elements are present, then not only are the outputs compared but the logic cones driving each pair of corresponding registers is checked individually. Since the RTL and netlist state elements directly correspond to each other, this check guarantees that the netlist is truly an implementation of the RTL.

As depicted in Fig. 1, the *key points,* or significant points used by the FEV tool, are composed of the input-output interfaces and the internal state elements. The main points of comparison are the state elements {R1, r1} and {R2, r2} and the outputs X and Y. As long as each of these points can be shown to have logically equivalent combinational driving cones in both models, one can confidently say that the models are equivalent.

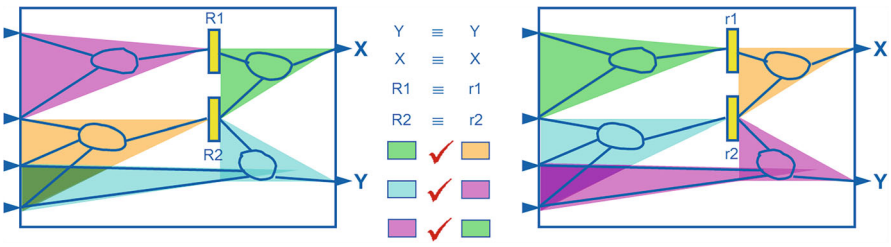


Fig. 1 Key point mapping in combinational equivalence checking

One can probably think of some major limitations of this type of equivalence. For example, what happens if the implementation needs to recode a finite state machine (FSM), using a different set of intermediate states to ultimately generate the same output? The application of combinational equivalence to FSMs is limited to comparing models that have equivalent sets of state elements. If this condition is not met, the combinational FEV tool will report the two models as non-equivalent. Modern RTL-netlist FEV tools do have some clever optimizations for very simple cases of non-state-matching, such as flop replication or constant propagation. But in general, the state-preserving requirement is a stringent one, and there are many FEV scenarios in which it must be relaxed for effective verification.

Sequential Equivalence

Sequential equivalence is also referred to as *cycle-accurate equivalence* by some vendors. With sequential FEV tools, one shall ask the question of whether two models will ultimately generate the same outputs at the same times based on an equivalent set of inputs, without requiring that internal state elements fully correspond. Thus, sequential FEV tools can handle cases like the recoded FSM described above.

In general terms, sequential FEV can be considered equivalent to the FPV, with the SPEC being equivalent to a large reference model. Instead of asking the question, "Is the property *RTL model output* == *reference model output* always true?", let's ask the similar question "Is the RTL model always equivalent to the reference model?" In fact, because good commercial sequential FEV tools have only appeared on the market very recently, it was common for many years for engineers to use FPV tools as simple sequential FEV tools in this way. They would create an enclosing module with both the SPEC and IMP model inside, create assertions that the outputs are equivalent, and run a standard FPV tool. However, now that there are tools available which specialize in sequential FEV, with the availability of a reference model which is complete enough that can just check equivalence instead of writing properties, it is usually better to use sequential FEV, which can take advantage of engine improvements targeted at equivalence checking.

Unlike combinational checkers, sequential equivalence checkers can verify designs with common functionality despite differences in:

- State representation
- Pipeline depth
- Interface timing and protocols
- Resource scheduling and allocation
- Differences in data types
- Clock gating and other power-saving features

For example, when using sequential FEV, one of the designs under comparison might be a pipelined version of the other. One could prove the equivalence of the

pure behavioral model of RTL to the complete pipelined implementation and satisfy that they do indeed implement equivalent functionality. One common application of this methodology is during the equivalence check of an unpipelined and pipelined model in RTL. In the simplest case of comparison, one can assume that the same input patterns are applied to the corresponding inputs of both the designs, and the outputs are compared with some known delay after the RTL pipeline is completely filled.

Transaction-Based Equivalence

This is another variant of sequential FEV. In some cases, one model will be highly abstract compared to the other, and its outputs will generally not match cycle-by-cycle except for certain well-defined transactions. This needs a looser notion of equivalence, based on checking that legal transactions are handled equivalently. The equivalence can be after a fixed number of RTL clock cycles, which could signify the transaction, or a completely unpredictable number based on the transaction completion. Hence, this notion is more generic and is a superset of all the equivalence modes: models which are not combinatorial or sequentially equivalent may still be able to demonstrate transaction-based equivalence.

Figure 2 illustrates a high-level view of a transaction-based equivalence check. To further clarify what FEV is doing, let's examine a conceptual view of the "transaction space" of a typical design.

This notion of equivalence is quite general and encompasses the previous notions of equivalence described in this section. This model of equivalence:

- Does not assume that the amount of time (or the number of state transitions) required to complete one transaction will be the same for the RTL and the system level design (SLM)

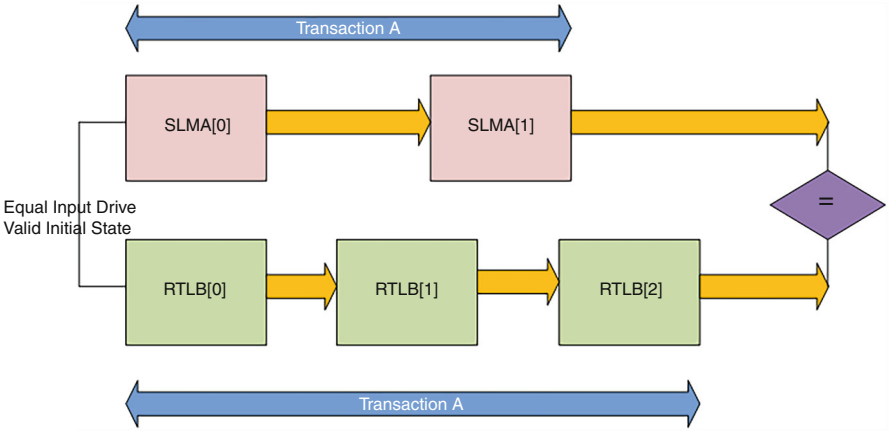


Fig. 2 Transaction-based equivalence check

- Denotes the end of a transaction by either a fixed number of RTL cycles or some kind of "data ready" handshaking protocol
- Assumes the user is able to specify a set of initial states (a partial assignment of values to the state-holding elements) for both the RTL and system level design, which represents the model state at the start of a valid transaction
- Assumes the user is able to specify a set of states for both designs that correspond to the end of a transaction

At the end of a transaction, the outputs of the two designs are compared, and a check is made to ensure the ending state of both designs still satisfies the state invariants and constraints.

Most performance optimizations would make the design retain the same functionality but wouldn't necessarily have a constant or definite relationship with the earlier implementation. In such cases, even a sequential equivalence check might not be a correct option to choose; the transaction-based equivalence should definitely help.

Verification Methodology

1. SystemC/C++ front-end compilation
   Specifications written in high level languages such as C/C++ and SystemC are used to test the functionality of complex algorithms. These specifications are exhaustively tested against standard WHQL and other standard algorithmic references and are considered golden w.r.t architecture specifications. Ideally, one would want to compare these golden specs directly with RTL to achieve confidence in the RTL design once the complete data space coverage is available. However, compiling a C++ specification in a formal tool can present a varying range of ease from straightforward compilation to almost exposing never converging data structures that pose an impossible task based on the how the design has been written and capability of the FV tool.

   The high-level code, agnostic to the formal verification requirements, tends to have a lot of features that are not formal friendly. The coding standards for these kinds of simulation code would aim in maximizing the efficiency and speed, which aren't the correct fit for the FV compatibility. Also, every company has a set of optimized and customized libraries which are essential in writing the golden specs. It would be challenging if the constructs used in these libraries are not supported by formal verification tool. Some examples could be extensive use of pointes, use of new and delete constructs (in context of dynamic memory allocation), extensive use of STLs, string functions (mostly used in test generation code), etc.

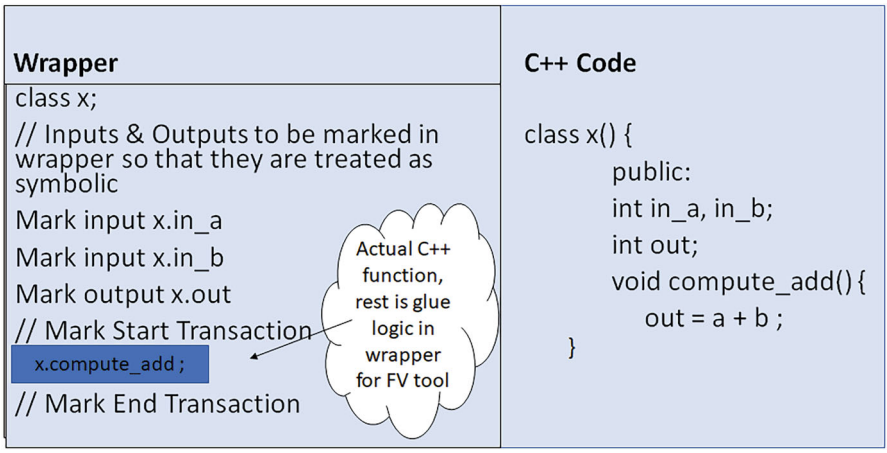
   In most of the cases, the problem may be due to inherent style and comfort level of the designer writing the code and it may be entirely possible to avoid such constructs by rewriting this in a formal-friendly way. However, the legacy code involved may have many such deep rooted constructs which would be impossible/impractical to rectify and rewrite. Handling these complexities

requires a crisp understanding of genuine requirements of the designers and the limitations of the formal verification tool.

Several tools have varied capabilities in parsing and supporting complex structures and libraries. Till about 5–6 years ago, the industry standard tools could only support C++/SystemC written with the intent of being consumed by a FV tool which implied that there was very none or limited support for dynamic memory allocation which disallowed users to use STLs, boost, pointers, etc. in their code. This limited the usage of these tools to only those complex algorithms where putting in the effort of writing a formal-friendly C++ code had good return on investments.

Even today, open source tools support only a limited subset of C++ for compilation. However, the industry standard tools have realized the potential of being able to consume a much wider range of C++. Available tools in the market today have a logic to parse some very involved C++11, C++14, and some C++17 constructs. They are able to support BitVector libraries and complex pointer logic. Dynamic memory allocation such as in STLs can also be supported to some extent, and the work is ongoing in almost all industry standard tools to enhance their front end. There exists decent support of STL in some tools, and hence most of the software optimized specification can also be consumed by the tools to start the equivalence verification. Classes, inheritance, typecasting, and virtual functions are also being supported. Recursive and looping constructs are consumed as long as the unrolling limit is reasonable. Else, tools offer ways for verification engineers to convey to the tool upper limits on the loops especially in case of nested loops or for functions. Some tools can consume SystemC and hence can work well with the C2RTL generation tools to verify the conversion correctness to RTL.

2. Wrapper around C++/SystemC model
   One of the prime requirements to enable C to RTL verification is that the interface boundary of the algorithm to be tested should match. Usually, in case



of simulation models, this boundary does not match as the designers are mostly focused on unit level interface matching and do not care about the algorithmic boundary. Hence most FV tools provide a standard way of writing a wrapper over the C++ code. This glue logic would clearly convey to the tool the inputs and outputs which need to be treated as symbolic. Also, it clearly identifies the C++ function which is to be tested using transactional equivalence.

The specification in C will be converted to a suitable format for the tool to process the design and start the verification activity. Some tools convert into a form of a binary decision diagram (BDD) and some prefer to create a data flow graph (DFG) .

3. Linting checks on C++/SystemC
   Many static checks can be run on the database that can be created from the compilation. The static checks can range from pointer checks, loop limits, typecasting checks, to range overflow checks. There are other linting checks also possible in C++ such as standard static assertions which flag the program crashes, lead to undefined behavior (such as out-of-bound array accesses, max iteration checks, division by zero checks, check whether value of second operator in a shift is within the bit width of the first operand, and so on). There are usually automatically checked by all tools in the compilation phase itself.
4. RTL front-end compilation
   There are many Verilog/SV/VHDL front-end parsers and compilers in the market that also translates the implementation design into a BDD/DFG.
5. Linting checks in RTL
   Multiple static RTL checks like multiple driver assignment, combo loop assessment, and undriven checks are possible on the implementation RTL. There are many static checks possible beyond the basic LINT checks and depends on the strong front-end compiler deployed.
6. C++ – RTL mapping
   Creating the verification problem needs a mapping of the two designs to be verified. The mapping can mean creating points of equivalent drives across various interfaces on the design under consideration. Many tools support mapping by name and hence can map the equivalent points across spec and implementation. Depending on the methods of solving the problem either running some sample dynamic simulations or semi symbolic checks or full blow symbolic verification, these mappings help in constraining the problem. There are various kinds of mappings which may be required such as the following:
   (a) Primary inputs: Either “map-by-name” or explicit mapping the signals between the wrapper on C module and primary inputs of RTL.
   (b) Primary outputs: Again, these could be “map-by-name” or explicit mapping the primary outputs.
   (c) Undrivens: Formulating the full problem for analysis needs driving all signals and some cases would need explicitly driving some signals with X/Z/discrete values.
   (d) Cutpoints: At times, there are signals in RTL which are not present as input in C++/SystemC code. They may be instantiated in C++ as global/static

variables or may be getting initialized in constructor class. For such signals, cutpoints are used to make the signals symbolic at a particular clock cycle. Then, they are mapped against the appropriate RTL signals. Marking a signal as a cutpoint makes the FV tool treat it as a symbolic value. Hence, the onus is on formal verification engineer to ensure that if there are any constraints on the cutpoint signal based on bit widths on any fixed value, it needs to be explicitly communicated to the FV tool. Cutpoints are also a very useful mechanism to aid in convergence by enabling divide and conquer discussed later in this chapter.

(e) Blackboxed modules: Blackboxing a module or some part of the algorithm needs removing that logic from cone of logic of the FV tool. Hence, the inputs of the blackboxed module are the points of the checks which need to be verified for equivalence. If inputs are proven, outputs of blackboxed module are assumed to be correct under the assumption that blackboxed module behaves in the same way in both HLS and RTL

(f) Undefined reg initial values
In RTL design, for resettable flops, the states are known. However, for non-resettable flops, initially as an overconstraint one might want to fix it to some initial value. That helps avoiding spurious counterexamples especially in cases where there is feedback in the circuit involving non-resettable flops.

7. Constraints on inputs
One of the axioms of formal world is that it considers the entire state space unless constrained. So, the onus is on the verification engineer to constraint the environment depending on actual design requirement such that there are no spurious counterexamples during equivalence checking.

(a) Signed/unsigned extension of input mappings
One common pitfall of mapping C++ variables with RTL signals is that if sign extension is not taken into account, the tool would map the exact bit width of RTL with the bit width of C++ and assign the higher bits of C++ to symbolic values rather than sign extending them. This leads to spurious counterexamples. One another common pitfall is that in RTL a 1 bit enable/hold signal may be used as a bool in C++. Verification engineers often map the two signals without realizing that the bool signal in C++ compiler is often treated as either 8 bit or like int 32 bit signal, hence upper bits also need to be appropriately constrained to get desired result.

(b) Modify implicit mapping
In some cases, especially in RTL2RTL, designers adopt map by name strategy to map all primary inputs and outputs to save time. In those cases, it is important to ensure that some signals may be required to behave in a different manner even though they may have same name. Such signals need to be unmapped and then explicitly constrained for the FV tool. For example, in clock gating, the enable in one RTL needs to be high while it needs to be low in second design to ensure correct clock gating verification.

(c) Latency
C++ is untimed. A wrapper written on top of it, if written in SystemC, often is treated as a 1 cycle delay. However, RTL can have variable latency. That latency needs to be provided in the equivalence either as a fixed value or in terms of constraints dependent on some signal (such as output_dv, etc).

(d) Throughput
In a stable state, one should be able to determine the frequency of incoming data such that design is able to output data at the same frequency. There may be a situation where to avoid interaction bugs, the pipeline only accepts one input every n number of cycles; this constraint needs to be provided to avoid spurious counterexamples.

(e) Clock, reset, and env constraints
The clock signals, behavior of resets, behavior of non-resettable flops in case of some designs, and other environment constraints which are part of peripheral logic driving the datapath algorithm such as enable signals, hold signals, dv, etc. need to be defined. To cite an example, in case of execution unit, one needs to clearly state that only one operation can be done at a time.

(f) Constants, tie offs
There may be additional signals driving the RTL circuit which do not have any impact on actual algorithmic implementation or which need to be driven to a constant value. This is true in case of C++ as well where they may be if-then-else conditions which need to be driven in a certain manner to drive the correct logic. For example, let us suppose code in C++ has an if statement to drive either a multiplication algorithm or an FMA algorithm in a single function. To prove this function against an FMA RTL, one needs to tie off the if condition in C++ to a fixed value driving FMA path. These tie offs are also useful to constrain the environment in case of huge designs with various paths with aim of achieving convergence using case splitting which would be elaborated later.

(g) Transactors/BFMs/master mode abstractions/checker BFMs
These are specific transactors for formal environment. Modeling extra logic to align the signal level details between HLM and RTL implementation. It is strongly advisable to mimic the logic as BFM/transactors. For example, a signal pre-dv is defined in RTL which comes a signal before the dv signal and related data input in the implementation, while the specification in C++ would define the dv signal at the interface. It is recommended to write a transactor to model this instead of complicating the assumptions. There would be even more complex relations across various signals and hence transactors/abstraction models are preferred to model these relations. Transactors can also be used on the output interfaces to model such complex relations on the output side similar to the input interface assumptions and are called as checker abstractions/BFMs.

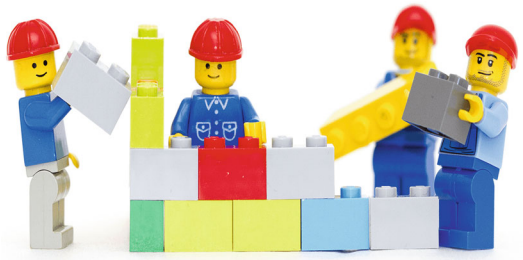
8. Blackboxing
   There may be modules in both C++ and RTL which may be either non-formal-friendly or not required for actual algorithm and would hamper convergence efforts. Such modules or functions can be chosen to be blackboxed in most formal verification equivalence tools using ignore functions or blackbox commands. Blackboxing modules after proving them is also a neat method of achieving convergence on a huge design which would be explained in this chapter.
9. Assertions for proofs
   The final check on equivalence is that given the mappings between inputs and outputs and the correct set of constrained environments, the final outputs should match between the spec and the imp at the appropriate latency mentioned. The verification engineer can also write additional lemmas for sanity of the RTL code such as to check the expected outcome on output control signals such as output_dv, output_hold, and flags.

   There may be bypass cases not covered in C++ which can be simply checked in verification tool with assertions such as if certain bypass condition in RTL is true, output is equal to input, etc.

   In addition, once the setup is ready, properties can be written only on the C++ as well to ensure design intent. For example, if an architect expects that output of an algorithm can never be negative or any intermediate signal is always within a certain range, then properties can be written only on C++ to validate the desired intent. In case of fixed precision numbers, properties can be written to check the error range with floating point numbers as well.
10. Verification
    (a) Dynamic quick checks (2 valued vs 3 valued)
        All formal verification tools are equipped with a constrained random simulator to quickly ascertain the sanity of both C++ and RTL design and ensure that there are no simple bugs in the design. The simulator is run even before formal model is compiled. This gives formal tools an edge over simulation tools in terms of speed and efficiency. The tools can run 2 valued simulation or 3 values simulation while the formal proof is being done.
    (b) Full proofs
        FV tools are equipped with various engines. The brute force resorts to using SAT/SMT solvers and BDD-based engines. For more complex algorithms, most tools have specific engines to deal with different algorithmic variations such as bit wise operations, multiplication, shift operations, etc. These engines are equipped with various word level solvers to cater to specific problems.

        For example, there may be multiplier implemented in RTL with different optimized algorithms such as booth multiplier, partial products, radix multiplication, and Wallace tree multiplication.

11. Counter examples, debug
    FV is known to provide the shortest counterexample for the assertion. It is prudent to mark the signals which are not in the cone of logic of the assertion as either X or any other symbolic value. Most tools have a good and mature debug environment for RTL, C++ debug environment is being constantly developed by the tools owning to competing industry standards and popularity and ROI of this equivalence technique.
12. Convergence
    (a) Modularity



    Modularity is the ideal way of achieving end-to-end convergence by guiding the tool to identify intermediate points between SPEC and IMP to match and then use those intermediate points to prove further logic. This divide and conquer technology is the most commonly used convergence technique as there is no additional effort required except understanding the modularity of the design structs. However, this is contingent on finding the intermediate points. If the designs are such that due to the huge difference in their abstraction levels, there can be no intermediate points to match, then this is of no help. For example, for a multiplier, it may be implemented using a "*" operator in C++, but it may be a Wallace tree implementation in RTL. In such a scenario, there are no intermediate points to match.

    (b) Blackboxing
    Sometimes, a module in C++ may be non-formal-friendly such as log function. In that case, it is advisable to blackbox the function in C++ and equivalent logic in RTL. Blackboxing is also important to prove a module first, then blackbox the module and prove assuming that blackboxed module is equivalent. This involves first proving that inputs to the blackboxed module match in both SPEC and IMP. Assuming given inputs are same, outputs of the blackboxed module is also same.

    (c) Case splits
    Many algorithms have multiple paths decided by various parameters or input signals. For example, in an address translation algorithm, there may be multiple surface formats, depending on the surface format; there may

be further surface types, pixel modes, etc. In encryption algorithms, or in matrix multiplication algorithms, there may be various bit widths deciding the course of the algorithm in a single function. In a blend algorithm, there may be as many as 30 blending modes which parse through the same function with caveats depending on the mode. In compression algorithms, there may be multiple formats and further signals dependent on those formats driving the logic. In floating point execution, the way normal numbers are treated is entirely different from how bypass cases such as INF, NAN, etc. are treated. Aiming to achieve end-to-end convergence for such complex designs with varied paths is almost impossible due to huge state space. In such cases, case splitting is a commonly used technique. Most tools are automatically able to create case splits and further hierarchical case splits based on user input to run each case separately in parallel. For such automatic case splits, tools create a top level proof to ensure that all cases together call for complete coverage of the design.

(d) Cutpoints

We discussed cutpoints in context of initial mappings earlier in the chapter when intermediate signals of C++ (usually global variable) is being mapped. Adding cutpoints is a very powerful tool in the convergence techniques arsenal as it has a potential to substantially reduce the size of state space of the combined SPEC and IMP design. Most tools have standardized ways of adding cutpoints.

Adding a cutpoint in either SPEC or IMP essentially means asking the tool to ignore the fanin of that signal and consider that signal to be symbolic. One example to elaborate is in the diagram below, Fig. 3; consider a serial FMA circuit. The logic surrounding the multiplication is intense and then there is a logic and probably latency associated with addition in RTL circuit. If proved that the multiplication output of both circuits is same for stage 0, one can add cutpoints in both SPEC and IMP at multiplication output to make it symbolic. This reduces the further proof to just proving that addition logic in both circuits is now same substantially reducing the complexity of the problem.

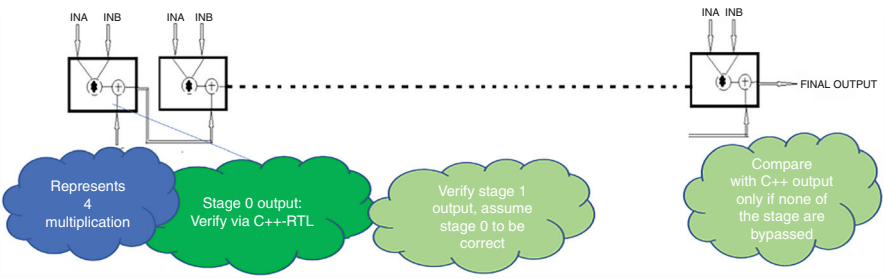


Fig. 3 Serial FMA circuit

One problem associated with using cutpoints is that the signal is now driven as being symbolic, so if there were any constraints to be associated with this signal which are necessary, it is needed to now separately constraint them to avoid spurious examples. For example, if due to driving logic, let's say that the signal always had upper 3 bits as zero (RTL/C++ designer chose extra bit widths to accommodate for future projects), then this needs to explicitly marked this constraint.

Finding such constraints is difficult as designs are also often not aware of how internal signals should be constrained. One workaround for such cases is to add cutpoint only in either SPEC or IMP, whosoever logic is more involved, that way one gets reduced state space from the circuit where the cutpoints are added to get correct constraints from the circuit where the cutpoints weren't present.

Cutpoints are also useful to enable blackboxing some functions which are non-formal-friendly such as log in C++; it is recommended to first prove that input to log function matches some intermediate signal in RTL. Then, add a cutpoint at the output of log function and map it to the output of log look up table (standard) of RTL. This helps to bypass log and still claim end-to-end convergence.

(e) Dynamic weakening
This technique helps to guide the tool regarding the signals which are not expected to be in cone of logic and can be ignored. This speeds up the process of convergence.

(f) Assume guarantee
Assume guarantee is a technique to prove intermediate points in the 2 designs and assume them for further proofs. Adding all assertions as assumptions can lead to a huge state space preventing convergence in some cases for simple proofs. Hence, most tools have advanced assume guarantee techniques to choose logic to prove through this technique and a choice to choose which assertions must be treated as assumes.

Convergence: Once all failures are resolved, one may face convergence issues because of huge gate count of design. This requires specialized treatment using path analysis, cutpoints, etc., and FV team adds those features to close the verification process. Formal guarantees 100% coverage for datapath algorithm which attracts all people involved.

13. Bug hunting
When it is hard to achieve 100% convergence such as for very complex designs such as compression, then it is recommended to run formal in bug hunting mode. Bug hunting involves taking a dynamic symbolic simulation run to bring the design state to a certain point and then start formal proof from that state. This allows for some deeper bound testing.
14. Coverage
    (a) Case splits coverage
    (b) Data space coverage
    (c) Assumption coverage
    (d) Interesting data cross coverage

15. Regressions
    Once an algorithm is proven, it is important to enable regression setup such that for every minor change in C++/RTL, testing is automated. This allows the user to only look at the setup again in case there is a failure in future revisions.

Using Design Exercise for Datapath Designs

General design exploration aims at exploring the design through some cover properties and playing with the waveforms and properties to get a preliminary confidence on the functionality of the design. The process starts from simple wiggling to exploring complex behaviors formally. This allows designers to create a simple formal test environment and helps reduce design development and debug effort. There are various tools in the industry which allow creating such formal test environment without writing too many properties. Design exercise is recommended at a preliminary stage of design development and want to quickly gain confidence in its functionality, with a relatively lightweight verification effort. However, it requires a clear-cut plan on what to check and when to feel that design exercise is done. For example, when dealing with a state machine design in control path, the most obvious goal is to check that every legal state can be exercised. It is also recommended to look at specification document and make sure that all paths are asserted and all required behaviors are exercised. In control path, one can decide on exit criteria on the basis of coverage report or time limit. In datapath designs, design exercise is used in a slightly different context. The aim is still to check functionality of the design in early design development, but it differs from control path in the sense that datapath eventually confirms 100% coverage at the end of verification. The effort involved would eventually be same as full verification except that one can quickly find bugs and ensure complete dependence on formal and rule out/reduce simulation cycles. Standard validation involves waiting for a Golden C++ Specification to be fully developed and tested to establish its validity using simulation. After that, C++ vs RTL formal verification is done using a datapath FV tool to find bugs and prove that the RTL algorithm implementation is indeed correct. In design exercise, it is strongly recommended to validate the behaviors even before the simulation environment is fully up and authenticity of the C++ specification has been established. The RTL is also, of course, in development stage.

	Golden C++ – RTL Verification	Design Exercise
Stage of design development	C++ is tested, RTL is preliminary or mature	C++ and RTL both are preliminary
Coverage	Requires 100% coverage to claim confidence	Aim is to find initial bugs to ensure confidence
Verification effort	Requires designers help to write constraints, FV team help to ensure convergence	Requires designers help to write constraints, convergence is required much later when all bugs have been fixed and RTL is mature

The design exercise can be broadly categorized in following steps:

Compiling C++ in datapath tool first time: Compiling a C++ specification in a formal tool can present a varying range of ease from straightforward compilation to almost exposing never converging data structures that pose an impossible task based on the how the design has been written and capability of FV tool.
Set up formal environment: This requires setting up mappings, constraints, etc. There may not be direct mappings, so it requires some expertise to define and add cutpoints, etc.
Design exercise per feature: The design exploration maximizes its return on investment if targeted one feature at a time. Once the basic setup is ready, the onus is on designers to debug failures and resolve failures. If there is a gap in their understanding with respect to intent of the algorithm, prompt help from the architect can be taken; however usually architecture specifications suffice. Many corner cases are caught this way as quick failures with shortest path to counterexamples are achieved rather than writing own cover points and testcases in simulation. During each repeat, either C++ is changed or RTL is changed. Changes in C++ does not affect the compilation in general as all the issues related to header files linking or dynamic memory allocations, etc. have already been resolved and can simply be carried forward with ease.
Convergence: Once all failures are resolved, one may face convergence issues because of huge gate count of design. This requires specialized treatment using path analysis, cutpoints, etc. and FV team adds those features to close the verification process. Formal guarantees 100% coverage for datapath algorithm which attracts all people involved (Fig. 4).

The use case of this methodology covers four main areas:

Check functionality of big feature with an established algorithm in architecture specification.
Check algorithms which have interesting corner cases and have potential to be missed by both C++ designer and RTL designer.
Incremental changes for new features/enhancements.
Design optimizations for the existing algorithms

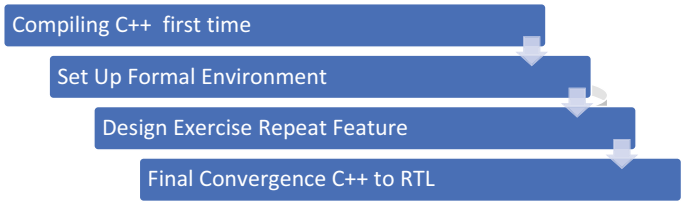


Fig. 4 Design exercise methodology

However, there are also some limitations:

It requires collaboration so designers need to be convinced of the importance of FV C++ needs to compile else it is a no-show.

It only spits one counterexample at a time rather than pointing all issues at once, which is the fundamental premise of formal. However, sometimes designers who have been dealing with only simulation may find it to be a limitation.

Advanced Datapath Verification

Managing Inconclusive Proofs

Given that there are infinitely many logic designs that perform the same function but a finite number of automated formal verification solvers and strategies, inconclusive proofs are, at some level of design complexity, inevitable. Designs that once were conclusive can easily fail to be so once RTL engineers invent novel ways of optimizing hardware performance. Quality RTL development and improvement must go hand in hand with encountering and managing inconclusive equivalency checking proofs.

While there are standard techniques for attempting to overcome these challenges; through case splitting, black-boxing, cutpoints, or using different solve scripts, they provide no guarantee of success. *Resolving* inconclusive proofs may require these techniques but *managing* inconclusive proofs requires a different approach. Simulation-based verification gains confidence in its results with every passing computation; equivalency checking may not, as techniques such as case splitting will never achieve proof (unless the input domain is trivial in size). How can managers force and guarantee progress on inconclusive proofs?

While exhaustive simulation at a bit level typically exceeds years for any sufficiently valuable datapath design, an approach that reasons at an operator level, i.e., integer operations, exponentially reduces the problem. Moreover, reasoning at an operator level matches how RTL designers reason about their implementations. While case splitting *may* resolve inconclusiveness, the only guaranteed approach is for the formal verification engineer (verifier) to dive into the two designs and work their way toward understanding *why* the two designs are equivalent. The practical way to do this is to build a set of equivalent RTLs or system level models from the specification design $\boldsymbol{S}$ and implementation $\boldsymbol{I}$:

$$S \iff S_1 \iff S_2 \ldots \iff S_n \quad I_m \iff \ldots I_2 \iff I_1 \iff I \qquad (1)$$

where $A \iff B$ indicates that a conclusive equivalency checker proof exists between designs A and B. These intermediate designs should be created to build a provable equivalency path between $\boldsymbol{S}$ and $\boldsymbol{I}$. This may require unoptimizing RTL, restructuring submodules to make internal boundaries equivalent, changing bit level

operators into their word level equivalents, etc. This level of white box knowledge will invariably require input from the creators of $\boldsymbol{S}$ and $\boldsymbol{I}$ or understanding the algorithms that produced them if auto generated.

This *waterfalling* approach is intensive but it will provide deep design understanding and force the verifier to understand which nature of design rewrites are provable by the equivalency checker tool. Note that progress is made with every step and even duration of subtasks of this process can be estimated. Ultimately a point will be reached of the form:

$$S \iff S_1 \leftrightarrow S_2 \cdots \leftrightarrow S_{n\prime} \iff I_{m\prime} \leftrightarrow \ldots I_2 \iff I_1 \iff I \qquad (2)$$

where in addition to the tool proofs of equivalence, $\iff$, certain links in the chain of designs, denoted $\leftrightarrow$, are considered "obvious" transformation according to the verifier but are not tool proven. Now Eq. 2 embodies the verifier's chain of reasoning as to why the designs are equivalent and answers the managerial question *Why is this a hard verification problem? How different are these designs?*

A full proof now requires resolving these human "obvious" tool "non obvious" transformations. Resolving this set of transformations will provide invaluable reusable strategies to the verifier and their team. These final transformations can be tackled in parallel and individually and independently tracked and resolved. The options:

- **Blackbox** – use cutpoints to isolate the differing parts of the designs and then apply case-splitting on that region.
- **Extreme Waterfalling** – introduce additional intermediate designs with even smaller axiomatic rewrites.
- **Human Sign Off** – formal verification is the act of risk mitigation not elimination, formal verification tools can return false positives. Time-bound verification effort may force the verifier to personally sign off on the correctness of a particular transformation.
- **Generate Supporting Evidence** – explore other methods to provide evidence, e.g., reducing the bit width of certain internal variables may provide evidence that the architecture of the design is correct (this can actually result in a full proof in certain cases (Shekhar et al. 2008)).

Extreme waterfalling forces the verifier to use equivalency checking tools as machine checking proof assistants, breaking down transformations into ever smaller axiomatic steps. This process crystallizes the difference between the verifier's definition of incremental and that implicit in the equivalence checking tool. This process requires a shift of granularity and a discovery of the multiple axiomatic steps typically taken in a human asserted transformation. As an example, consider this transformation for unsigned integers a, b, and c involving splitting the unsigned n-bit variable a into its constituent bits:

$$\overline{a} * b + a * c = \sum_{i=0}^{n-1} 2^{i} \left(\overline{a_i} * b + a_i * c\right) \tag{3}$$

Whether this transformation is or is not tool provable for a given tool and version is irrelevant, the verifier must be able to break this nature of transformation down into smaller axiomatic steps, e.g.:

$$\overline{a_{n-1:0}} * b + a * c = 2 * \left(\overline{a_{n-1:1}} * b + a_{n-1:1} * c\right) + \left(\overline{a_0} * b + a_0 * c\right) \tag{4}$$

Note that the right-hand side of Eq. 4 mixes bit and word-level representations of a and thus provides an intermediate design point for the transformation in Eq. 3. If Eq. 4 cannot be tool proven, this transformation can be further broken down into associativity and distributivity properties of multiplication and addition.

The crucial element is the generation of new intermediate steps, and these are the moments when those managing but not performing formal verification activities can provide invaluable direction.

In summary, waterfalling, creating multiple intermediate designs, provides a way to guarantee progress when formally verifying complex datapath designs. This forces intellectual understanding of the designs as well as the equivalency checking tools. This approach comes from basic manager questions, *Why is this hard? What are the differences between these designs? How to shrink this problem? Show the pain points in this verification. How can one overcome them?* These questions forcibly splinter the problem into its core constituent inconclusive proofs. Ultimately, they rest on the power of the equivalency tool to perform arbitrary bit width axiomatic rewrites. Once managers know the limits of the tool's capability in this area, they know what is and is not theoretically possible to prove by equivalency checking. This approach also removes the equivalency tool R&D from the critical path of any equivalency checking project. Formal verification teams must store these fundamental inconclusive proof problems, engage with the tool R&D on their resolution, and find solutions within the verification team before they become critical.

As formal verification teams working on equivalency checking evolve, they may even be able to get intermediate designs from the RTL or system level modelers themselves. Advanced RTL design teams may use waterfalling as part of their RTL design phase, shrinking design, and verification schedules.

While the equivalency tools certainly keep improving, waterfalling provides an approach to control, focus, and even schedule the work required to overcome highly complex inconclusive equivalence checking problems.

Accuracy Challenges

Formally verifying datapath designs naturally requires bit accurate specifications in order for proofs to be even phrased. The verifier will certainly perform component level verification. Beyond this however, they may ask or be asked questions

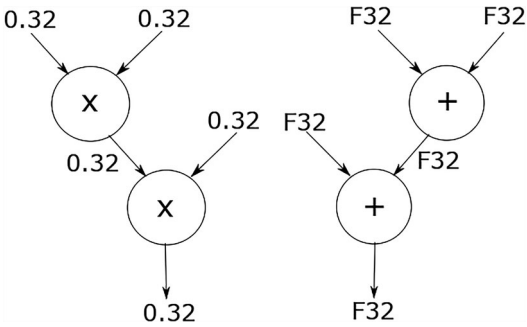


Fig. 5 Trivial fixed and floating point designs

Table 1 IEEE directed rounding modes

Rounding mode	Floating point output
Round to nearest, ties to even	Return the nearest of F1 and F2, if equidistant, return the "even" value
Round to nearest, ties away from zero	Return the nearest of F1 and F2, if equidistant, return value furthest from zero
Round toward zero	Return the value nearest to zero
Round to positive infinity	Return F1
Round to negative infinity	Return F2

regarding algorithm level accuracy. Consider the simplest of fixed point and floating point designs in Fig. 5.

The left-hand side design is fixed point with a 32-bit fractional datatype used throughout, denoted 0.32. The right-hand side design is floating point with a single precision IEEE 754 datatype used throughout, denoted F 32. The most accurate components that can be created would use *round to nearest, ties to even*, also known as correct rounded, for their rounding mode (a full explanation of rounding modes is provided in Table 1). However, it is worth noting the accuracy of these trivial designs when using the most accurate components.

For the fixed-point design, consider the following implementation result, where $\hat{\times}$ denotes the round to nearest, ties to even fixed-point multiplication operation:

$$\begin{aligned} \text{Infinitely Precise}: \quad & 0.1 \times 0.111\ldots111 \times 0.111\ldots111 \\ & = \tfrac{1}{2}\left(1 - 2^{-32}\right)^2 = \tfrac{1}{2} - 2^{-32} + 2^{-65} \end{aligned} \tag{5}$$

$$\begin{aligned} \text{Implementation}: \quad & \left(0.1 \hat{\times} 0.111\ldots111\right) \hat{\times} 0.111\ldots111 \\ & = 0.1 \hat{\times} 0.111\ldots111 = \tfrac{1}{2} \end{aligned} \tag{6}$$

Combining two such components has resulted in an error which is nearly 2^{-32}. The implementation returns 0.1, whereas a correct rounding of the infinitely precise result would return $0.011\ldots111$. As expected, combining two correctly rounded components does not result in a correctly rounded result.

For the floating-point design, consider the following implementation result, where $\hat{+}$ denotes the round to nearest, ties to even floating point addition operation:

$$\text{Infinitely Precise}: \quad 2^{24} + 1 + \left(-2^{24}\right) = 1 \tag{7}$$

$$\text{Implementation}: \quad \left(2^{24} \hat{+} 1\right) \hat{+} \left(-2^{24}\right) = 2^{24} \hat{+} \left(-2^{24}\right) = 0 \tag{8}$$

The correctly rounded components introduce a relative error of up to 2^{-23}. Combining two such components has resulted in a relative error of 1, with a result which is 100% error. Such cases of, so called, catastrophic cancelation render the output to have no accuracy whatsoever. Combining two correctly rounded components does not result in a correctly rounded result.

These are trivial designs which demonstrate that any combination, no matter how trivial, of correctly rounded components will not result in a correctly rounded result. This also implies that if there is a reference implementation composed of multiple correctly rounded components, this itself will not be a correct rounding of the infinitely precise result. Where one might see a reference implementation in double precision floating point, while it is more precise it can only be *a* reference, it is not *the* correctly rounded infinitely precise reference.

If the reference model is not a correct rounding of the infinitely precise result and the implementation deviates from the reference, implementations must be considered error tolerant. Every floating point algorithm is thus error tolerant. Any reasonably sized hardware forces intermediate signals to have reasonably sized bit widths which forces rounding error at every stage of a calculation, and thus the final result will never be a correct rounding. Finite precision implies application-level error.

This raises the question, if the reference models are not correct roundings, why are correctly rounded components necessary? The answer: **they are not**. The IEEE 754 standard defines five directed rounding modes, if an infinitely precise F is exactly representable in the output floating point format, then that value must be used as the final output. If, however, F resides between two floating point numbers:

$$F1 < F < F2 \tag{9}$$

Then which of the two outputs to be used is defined by the rounding mode as shown in Table 1.

One could ask which of these rounding modes should be used for the reference model or the implementation. Also note that each of these rounding modes will have a different hardware implementation cost. Moreover the two *round to nearest* modes should be more expensive to implement in hardware than the other three, as their worst case error is half that of the others. Each of these rounding modes chooses between $F1$ and $F2$, either is a legitimate directed rounding. These five rounding modes are all examples of a *faithful rounding* which is defined in Table 2:

Table 2 Faithful rounding

Rounding mode	Floating point output
Faithful rounding	Return infinitely precise value if representable, otherwise return either F1 or F2

Table 3 Floating point single precision multiplier hardware cost

Rounding mode	Delay (ns)	Area (μm^2)
Round to nearest, ties to even	1.23	8536
Round toward zero	1.02	8208
Faithful rounding	0.98	6057

A faithful rounding can return either of the neighboring representable floating point numbers. Every one of the five IEEE 754 directed roundings is a faithful rounding. Moreover, the worst-case error of any faithful rounding is equal to that of round toward zero, round to positive or negative infinity; so, algorithm correctness that is based upon these three rounding modes will also hold true for faithful rounding. Incredibly this slightly looser accuracy requirement can be exploited to significantly improve hardware implementations. An example of the potential hardware benefits of using different rounding modes for a single precision floating point multiplier can be found in Table 3 (these results come from (Drane and Jain 2012) which utilized Synopsys' Design Compiler and a TSMC 65lp library).

In this case, the faithfully rounded component is 21% faster and 29% smaller than the correctly rounded component.

The verifier of datapath intensive designs should not only need to know which accuracy or rounding modes are being used with any given design but should also question the necessity of any particular scheme. Given every design will not be a correct rounding of the infinitely precise result; questions should always be posed regarding the necessary accuracy and precisions of components.

Accuracy Optimized Component Verification

Components which are not correctly rounded but faithfully rounded or correct to a number of units in the last place may offer significant hardware benefits but present multiple verification challenges. These *accuracy optimized* components have no bit accurate specification. Given the inherent freedom built into faithful rounding, there can exist multiple designs which are all faithfully rounded but are all functionally different from each other. It is the RTL designer that specifies the bit accuracy of the design. They will endeavor to maximally exploit the accuracy freedom to optimize the hardware. The bit accurate functionality will only be known once the design space exploration is completed. Therefore, no independent bit accurate specification can be created. For design flows that require system level models as well as RTL, system level models will need to be generated directly from the RTL. Performing equivalency checking between the generated system level model and the RTL provides no evidence of correctness. This is in contrast to the standard design methodology where system level models are independently generated and thus

equivalency checking between the system model and the RTL provides significant evidence of correctness.

Therefore, in order for RTL designers to make use of the hardware benefits offered by accuracy optimized components, non-standard verification techniques need to be employed. Moreover, the only evidence that can be provided of correctness will come from these non-standard techniques. These techniques use equivalency checkers to perform *Datapath Property Checking*.

Proving Faithful Rounding

Given the multitude of designs which can exhibit faithful rounding, standard equivalency checking will not be able to prove faithful rounding. However, referring back to Eq. 9 and Table 1, $F1$ is always returned by round to negative infinity and $F2$ always returned by round to positive infinity, hence faithful rounding can be rephrased as:

Faithful rounding returns	either	the round to negative infinity
	or	the round to positive infinity result

(10)

Note that if the infinitely precise value is representable in the floating-point output format then both the round to negative and positive infinity results will be identical and the faithful rounding definition in 1.10 forces the faithful rounding to also be the infinitely precise value. This property can be used to setup an equivalency checking problem that proves faithful rounding; one can create the modules as in Fig. 6.

Reference designs for both round to negative infinity and round to positive infinity form one design (RTNI Ref and RTPI Ref, respectively), and both reference

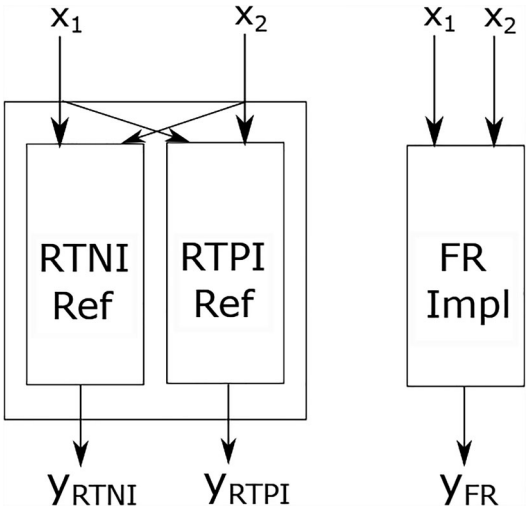


Fig. 6 Design setup for proving faithful rounding

values are returned (y_{RTNI} and y_{RTPI}, respectively). The implementation that is designed to be a faithful rounding (FR Impl) returns y_{FR}. The lemma that should then be given to the equivalency checker that proves faithful rounding is then in Eq. 11.

$$(y_{RTNI} == y_{FR}) \;||\; (y_{RTPI} == y_{FR}) \tag{11}$$

Equivalency checkers are able to prove such properties in practice; in (Drane and Jain 2012), a double precision floating point multiplier was proven to be faithfully rounded in under 2 hours on a 1.86 GHz Intel Xeon® machine.

Proving Monotonicity

While faithful rounding has no worse error than the set of directed rounding modes and can provide significant hardware benefits, it is not enough to prove only the faithful rounding property. Loosening the accuracy from directed rounding to a faithful rounding inadvertently runs the risk of violating unspoken assumptions about arithmetic components. Beyond proving the faithful rounding property, the verifier will have to consider all arithmetic dangers faithful rounding (or looser accuracy statements) may introduce, updating specifications and proving additional properties in the process.

Consider this trivial arithmetic property:

$$0 < a < b \Rightarrow \frac{1}{b} < \frac{1}{a} \tag{12}$$

Unlike the directed roundings, it is possible to build a faithfully rounded reciprocal function, *recipFR*, that violates this *monotonicity* property:

$$0 < a < b \; and \; recipFR(a) < recipFR(b) \tag{13}$$

As a precise example, consider:

$$F1 = \frac{1}{2} + 2^{-24} < \frac{1}{2 - 2 \times 2^{-23}} < \frac{1}{2 - 3 \times 2^{-23}} < \frac{1}{2} + 2^{-23} = F2 \tag{14}$$

Here the reciprocal of $2 - 2 \times 2^{-23}$ and $2 - 3 \times 2^{-23}$ lie between $F1$ and $F2$. By the definition of faithful rounding, one may choose:

$$a = 2 - 3 \times 2^{-23} \; and \; recipFR(a) = \frac{1}{2} + 2^{-24} = F1 \tag{15}$$

$$b = 2 - 2 \times 2^{-23} \; and \; recipFR(b) = \frac{1}{2} + 2^{-23} = F2 \tag{16}$$

These assignments result in the situation described in Eq. 13, and *recipFR* is an increasing function for these inputs. Faithful rounding allows for the creation of

reciprocal functions which *can* be an increasing function as opposed to its strictly monotonic decreasing infinitely precise counterpart.

While monotonicity may have never been considered, or even understood, prior to the use of faithfully rounded components, such questions will now need to be asked and answered. The property that should be checked by an equivalency checker is in Eq. 17.

$$recipFR\,(NEXT(x)) <= recipFR(x) \tag{17}$$

where *recipFR*(*NEXT* (*x*)) would be a design which first computes the adjacent more positive input to *x* and then computes the faithfully rounded reciprocal. Equivalency checkers are able to prove such properties in practice; in (Drane and Jain 2012), a single precision floating point reciprocal was proven to be monotonic in under 20 minutes on a 1.86 GHz Intel Xeon® machine.

Proving Commutativity

Unfortunately, faithful rounding can also violate a far more common property, commutativity. It is possible for the following to occur:

$$a \times b = b \times a \tag{18}$$

$$multFR\,(a, b) \neq multFR\,(b, a) \tag{19}$$

To see how this can happen, consider the following faithful rounding of an unsigned two-bit multiplication

$$x\,[1:0] \times y\,[1:0]$$

$$RTNI\left(\frac{x \times y}{4}\right) \leq multFR\,(x, y) \leq RTPI\left(\frac{x \times y}{4}\right)$$

$$\left\lfloor \frac{xy}{4} \right\rfloor \leq x\,[1]\,(y\,[1] + y\,[0]) \leq \left\lfloor \frac{xy+3}{4} \right\rfloor \tag{20}$$

Equation 20 can be trivially checked to hold for all possible values of $x[1:0]$ and $y[1:0]$. The bounds on *multFR*(x, y) are symmetric in x and y, but this particular faithful rounding implementation is not. In particular:

$$multFR\,(2, 3) = 2 \neq 1 = multFR\,(3, 2) \tag{21}$$

In fact, whenever multiplication is implemented with asymmetry between the inputs, such as in Booth encoded integer multipliers, there is a high risk that

faithfully rounded multiplication will in fact be non-commutative. In this case the property that must be checked by equivalency checkers is:

$$multFR(a, b) == multFR(b, a) \tag{22}$$

Equivalency checkers are able to prove such properties in practice; in (Drane and Jain 2012), a single precision 3 input floating point adder was proven to be commutative between its 3 inputs in 20 h on a 1.86 GHz Intel Xeon® machine.

In conclusion, when dealing with systems of datapath components, the verifier will be faced with the challenge of proving accuracy properties of the components as well as the system. RTL designers seeking to push the limits of hardware quality will consider non-standard component precision and accuracy. A common type of *accuracy optimized* components will use faithful rounding, due to its close relationship with IEEE 754 rounding modes and the hardware benefits associated with their usage. However, formally verifying faithfully rounded components requires non-standard use of equivalency checkers as well as addressing, typically non-verbalized, arithmetic properties such components may possess.

References

Drane T, Jain H (2012) Property checking of datapath using word-level formal equivalency tools. In: Design and Automation Conference (DAC) User Track

Shekhar N, Kalla P, Meredith MB, Enescu F (2008) Simulation bounds for equivalence verification of polynomial datapaths using finite ring algebra. IEEE Trans Very Large Scale Integr Systems 16(4):376–387

Verification of Arithmetic and Datapath Circuits with Symbolic Simulation

36

Roope Kaivola and John O'Leary

Contents

Introduction 1270
Symbolic Simulation 1270
Symbolic Simulation as Formal Verification 1271
Symbolic Simulation Among Formal Verification Methods 1272
Chapter Outline 1274
Simulation 1275
Booleans and Undefined Values 1275
Circuit Simulation and Undefined Values 1276
Mathematical Model of Circuit Simulation 1279
Circuit Properties 1281
Mathematical Model of Circuit Properties 1282
Symbolic Simulation 1283
Symbolic Computation 1283
Simulation with Symbolic Values 1284
Mathematical Model of Symbolic Simulation 1287
Practical Considerations 1288
Simulation Scope Control 1289
Property Triggers 1289
Scope Reduction by Triggers 1293
Reachable-State Invariants 1295

Intel provides these materials as-is, with no express or implied warranties. Intel processors might contain design defects or errors known as errata, which might cause the product to deviate from published specifications. Intel, Intel Core, Intel Atom, Pentium and Intel logo are trademarks of Intel Corporation. Other names and brands might be claimed as the property of others.

R. Kaivola (✉) · J. O'Leary (✉)
Core and Client Development Group, Intel Corporation, Hillsboro, OR, USA
e-mail: roope.k.kaivola@intel.com; john.w.oleary@intel.com

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_37

Complexity Management 1297
Simulation Complexity 1297
Complexity Analysis 1299
Weakening 1300
Verification Flow 1302
Arithmetic Circuits 1304
Direct Verification 1304
Floating-Point Operations 1306
Floating-Point Addition 1307
Integer Multiplication 1308
Floating-Point Multiplication and Fused Multiply-Add 1310
Floating-Point Division and Square Root 1311
Industrial Verification 1314
Related Work 1315
References 1317

Abstract

Symbolic simulation extends standard digital circuit simulation with symbolic representations of values, covering behaviors of a circuit for all possible instantiations of the symbolic values in a single simulation. Used as a formal verification method, symbolic simulation is algorithmically simple and intuitive, which enables precise analysis and fine-grained mitigation of computational complexity, allowing the method to handle circuits that are above the capacity of most standard formal model checking tools, as well as circuits with highly enmeshed pipelines. Symbolic simulation excels in verification of deep targeted properties of fixed-length pipelines, in particular arithmetic and other datapath circuits. It has been extensively applied in industrial verification of different kinds of integer and floating-point arithmetic circuits, such as adders, multipliers, and fused multiply-adders and dividers. Thanks to repeatable systematic verification strategies for complexity management and mitigation, such verification can be carried out routinely in development projects with rapidly evolving designs. Symbolic simulation is less optimal for verification of general consistency invariants and other inductive properties, as the method does not automate reachable-state analysis and requires a higher degree of human interaction and expertise than standard model checking tools.

Keywords

Simulation · Verification · Arithmetic · Datapath

Introduction

Symbolic Simulation

Digital circuit simulation is a standard tool in the arsenal of every working logic design and validation engineer. Symbolic simulation extends this technology with

the ability to carry out a simulation with sets of values in a single simulation trace, using symbolic representations. This chapter provides a pragmatic high-level introduction to symbolic simulation and its usage in formal verification of hardware designs, both in the conceptual level and in practice over the last decades. Its intended audiences are working validation or formal verification engineers and validation managers. The chapter also gives a mathematically precise outline of the foundations of the method.

A traditional circuit simulator takes as its inputs a circuit design and a stimulus trace assigning values to some or all input nodes of the circuit for given time periods. The tool simulates the circuit's behavior with the given stimulus for a required number of clock cycles and produces a simulation trace, assigning values consistent with the circuit structure and the stimulus to all the signals in the circuit and all the time points relevant to the trace. For validation purposes, the simulation trace may be connected to external checkers probing signals in the trace and observing that their values are consistent with some external notion of correctness. Alternatively, the circuit itself may contain embedded checkers, which are evaluated in the natural course of the simulation.

In a symbolic simulator, the input stimulus may contain *symbolic* variables in addition to the traditional concrete Boolean values 0 and 1. These symbolic variables are effectively names of values, denoting sets of possible concrete values. The values of the internal signals computed in the simulation are then structural logical expressions on the symbolic variables on the inputs. For example, in a bit-level symbolic simulator, a single symbolic variable a corresponds to the set of Boolean values consisting of both 0 and 1, and if stimulus to a symbolic simulation trace contains the variables a, b and c, the internal signals might carry values like $a \vee b$ or $a \vee (b \wedge \neg c)$. Section "Symbolic Simulation" provides more thorough examples of symbolic simulation.

A single symbolic simulation trace corresponds to a set of ordinary simulation traces, covering behaviors of the simulated circuit for all possible instantiations of the symbolic variables with concrete values. This universality connects symbolic simulation to formal verification.

This chapter focuses on discrete bit-level simulation of digital circuits with well-defined clocks using a zero-delay circuit model. In this model, all combinational logic gates are computed without any gate delay, and all sequential logic gates, i.e., flip-flops and latches, change values instantaneously on clock edges. The model is at the same time simple and strong enough to capture all logic-level correctness aspects of circuits.

Symbolic Simulation as Formal Verification

Used as a formal verification method, symbolic simulation excels in verification of deep targeted properties of fixed-length pipelines, typically of the transactional form:

stimulus A at time t is followed by response B at time $t + n$

It has a unique ability to carve out the circuit logic relevant to the progression of a pipeline while ignoring the rest of the circuit and other transactions in flight.

As the approach is conceptually simple and concrete, it gives the human verifier a fine-grained visibility into the progress of the computation during a verification task, enabling precise analysis and mitigation of computational complexity bottlenecks. Because of these advantages, symbolic simulation can routinely handle circuits that are beyond the capacity of traditional model checkers, as well as circuits where the pipelines are too enmeshed to allow equivalence-based verification. On the other hand, symbolic simulation is less well suited for verification of general inductive safety properties, or circuits involving significant feedback loops, as it provides no automatic mechanism for reachable state analysis, a staple of the traditional formal model checking methods. Further, liveness properties cannot be directly addressed by symbolic simulation, as they require reasoning over infinite traces, and symbolic simulation inherently focuses on finite behaviors.

Symbolic simulation has been the primary vehicle for Intel® arithmetic formal verification for over 20 years. Most arithmetic execution engines of Intel processor designs over this period have been exhaustively verified using it. While the verification of the hardest operations requires great expertise and insight, the overwhelming majority of arithmetic operations commonly implemented in hardware can be fully verified with direct symbolic simulation with little user interaction.

Symbolic simulation for circuit verification was introduced in the form of *symbolic trajectory evaluation (STE)* by Seger and Bryant in (1990, 1995), and the related methodology was elaborated in more detail in Jones et al. (2001) and Seger et al. (2005). In this chapter, as well as in actual verification practice, the original theory of STE is extended by admitting a larger class of properties (Kaivola et al. 2009; O'Leary et al. 2013): all fixed time window properties built with arbitrary Boolean combinators are covered, whereas STE in its basic form restricts itself to verification of implications between conjunctions of direct signal references.

Symbolic Simulation Among Formal Verification Methods

Several other technologies address the same problem space as symbolic simulation:

1. Formal equivalence verification (Drane and Kiran Kumar 2022),
2. Formal property verification, as in traditional or bounded model checking (Vizel and Ivrii 2022; Seligman et al. 2015)
3. Theorem proving (Ray and Goel 2022; Harrison 2009)

All of these technologies are implemented in a variety of academic tools and the first two in several commercial ones as well.

Symbolic Simulation and Formal Equivalence Verification Among these three technologies, symbolic simulation and formal equivalence verification (FEV) are closest to each other in the sense that they often are used to target similar properties, the equivalence of behavior between an implementation and a reference model. Contrasting symbolic simulation with FEV, the authors' empirical experience has been that automated equivalence verification thrives on structural similarities between the implementation and specification and does not necessarily perform so well in the absence of those. Many industrial circuits are highly optimized, and the conceptual distance between the gate-level implementation and an abstract, transparent reference model is large. In the authors' experience, verification of such optimized hardware with FEV may require structural decomposition or the creation of an intermediate reference model that is related to the design through equivalence verification and then to a high-level reference model through inspection or theorem proving.

Symbolic Simulation and Formal Property Verification Comparing symbolic simulation with formal property verification (FPV), as in traditional model checking, the first thing to notice is that model checking can handle a much larger class of properties, as symbolic simulation is essentially restricted to the verification of invariants with finite time windows. Also, symbolic simulation is at its best when focusing on narrow, targeted, "dense" properties related to a small subset of behavior in a design, and FPV on looser, more random relations between signals that are not directly causally related, such as general consistency invariants. As a general rule of thumb, in the authors' experience, formal property verification, and especially SAT-based FPV, has been superior in control logic verification and symbolic simulation in data space.

The practice of verification by symbolic simulation is closest to bounded model checking (BMC), however, with important differences: BMC considers instances of a property in a time window up to a given bound starting from a properly initialized state of a system, whereas symbolic simulation focuses on one fixed time instance of a property in a trace starting from an unconstrained state. The focus on one fixed instance of a property is discussed in more detail in section "Circuit Properties". It is an advantage of symbolic simulation, as it allows users to analyze in detail what circuit logic is relevant and excise the rest. On the other hand, the unconstrained start state can make verification harder, since properties may spuriously fail on traces that are not properly initialized, and the user has to separately formulate and verify any required reachable-state invariants to exclude such spurious failures. Also, as a pragmatic difference, when bounded model checking falsifies a property, the user is typically given a single concrete counterexample, whereas in symbolic simulation, the user receives a symbolic expression characterizing the full failure condition, providing richer information as a basis for debug.

In the authors' experience, the largest single differentiating factor between symbolic simulation and either FEV or FPV is that symbolic simulation provides a

concrete path to computational complexity analysis and mitigation when automation fails and that the complexity can be related to gate-level and microarchitectural aspects of the design. When the automation provided by FEV or FPV works, and a tool is able to automatically either verify or falsify the property of interest, it provides great value with a small investment from the user's part. In contrast, symbolic simulation is clearly a specialist tool. However, when the fully automated tools fail to produce a result because of the complexity of the property or the circuit, they typically do not yield actionable information that would point the user toward a solution. On the other hand, the strong operational intuition of simulation makes the technique approachable to engineers without a traditional formal methods' background and allows them to analyze and solve complexity challenges that would otherwise block the work.

Symbolic Simulation and Theorem Proving Both symbolic simulation and theorem proving are human-driven, computer-assisted processes. The main difference is the extent and nature of the human intervention required. Traditional computer-assisted theorem proving both allows and often requires human guidance over the minutest details of the verification, and this guidance often reflects the flow of the theorem prover, not the circuit (cf Russinoff 1998, 2019). Symbolic simulation, on the other hand, allows users to let automation gloss over many design details, which is particularly useful in highly optimized bit-level designs. Also, most of the human guidance can be understood in terms of the flow of information in the circuit, making the technology more accessible than theorem proving. As discussed later in the chapter, symbolic simulation can be combined with theorem proving for problems that exceed the capacity of pure simulation. However, this reasoning typically takes place at a level of mathematical relations, not at the gate level.

Chapter Outline

The rest of the chapter is structured as follows. Section "Simulation" starts by discussing normal circuit simulation and the use of the undefined value X as an abstraction method to allow one simulation to cover simultaneously a set of many circuit behaviors. Section "Symbolic Simulation" introduces the concepts of symbolic variables and symbolic expressions and their use in simulation. In both sections "Simulation" and "Symbolic Simulation", the technical concepts are introduced in two ways: first, conceptually with the help of concrete running examples, and secondly, through a mathematically precise model. The former is more aimed for formal verification practitioners, to foster an intuitive understanding of the concepts to guide pragmatic verification work, and the latter for the research community, to allow precise comparison with other verification techniques.

After the basic concepts have been introduced, Section "Simulation Scope Control" examines simulation scope control, a key aspect of verification complexity containment, especially the technique of parametric substitutions and the role of reachable-state invariants in simulation scope reduction. Section "Complexity

Management" addresses complexity management and analysis in more detail and the role of the various weakening techniques in verification complexity reduction.

Moving on to more pragmatic aspects, Section "Verification Flow" describes the flow of verification work using symbolic simulation in practice, and Section "Arithmetic Circuits" catalogues case studies and specialized techniques for a variety of arithmetic circuits: simple arithmetic, floating-point operations, floating-point addition, integer multiplication, floating-point multiplication and fused multiply-add, and floating-point division and square root. Section "Industrial Verification" discusses aspects of large-scale industrial verification tasks and the role of repeatable, programmable verification strategies. Finally, Section "Related Work" outlines a brief history of symbolic simulation in circuit verification and contains a short literature survey.

Simulation

Booleans and Undefined Values

The most fundamental element of the circuit model used in this chapter is a combinational Boolean gate. In modelling these, the usual set of Booleans is extended with a special *undefined value* X, denoting lack of information (Definition 1). The value X means intuitively that it is not known whether a value is 0 or 1. As customary, the dual of X, the *overconstrained value* $\top$ is also added. However, $\top$ is rarely used in verification practice, and we do not discuss it further.

Note that the value X is a modeling abstraction. It refers to knowledge about a signal value, not to any electrical phenomena that would correspond to a signal value distinct from 0 or 1. At an underlying concrete level, every signal value at every time in an actual circuit behavior is expected to be strictly Boolean, either 0 or 1. Common Boolean operators can be naturally extended to cover X values as depicted in Fig. 1, reflecting the intuition that X denotes an undefined or don't-know value. Such extensions are called *monotonic* (Definition 2). The same concept is known as *X-pessimism* in digital circuit literature.

The key feature of the X-abstraction and monotonic extensions is that if a function is computed using its monotonic extension for an argument list that contains X's and the result is a Boolean value of 0 or 1, then the argument X's can be replaced with any combination of 0s and 1s, and the result remains the same

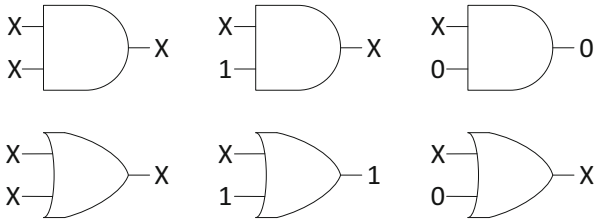


Fig. 1 Boolean operators and the undefined value X

(Lemma 1). This gives us the ability to compute the function only once and establish the result for multiple different concrete arguments. For example, from the single computation $X \wedge 0 = 0$, it can be concluded that both $0 \wedge 0 = 0$ and $1 \wedge 0 = 0$.

Definition 1. We write $\mathbb{B}$ for the set of Booleans $\{0, 1\}$. Let $\mathbb{X}$ be the set $\{0, 1, X, \top\}$ extending $\mathbb{B}$ by the *undefined value* X and the *overconstrained value* $\top$. We define the relation $\sqsubseteq \subset \mathbb{X} \times \mathbb{X}$ as the minimal relation that satisfies $X \sqsubseteq b$, $b \sqsubseteq b$, and $b \sqsubseteq \top$ for all $b \in \mathbb{X}$. If $b_x \sqsubseteq b$, we say that b_x *abstracts* b. For $b, b' \in \mathbb{X}^k$, we write $b \sqsubseteq b'$ if $b_i \sqsubseteq b'_i$ for all i, where $b = (b_1, \ldots, b_k)$ and $b' = (b'_1, \ldots, b'_k)$.

For any sets $A_1, \ldots, A_n$ and functions $f, f' : A_1 \to \ldots (A_n \to \mathbb{X})$, we write $f \sqsubseteq f'$ if $f(a_1) \ldots (a_n) \sqsubseteq f'(a_1) \ldots (a_n)$ for all $(a_1, \ldots, a_n) \in A_1 \times \ldots \times A_n$.

Definition 2. Let f_B be a function $f_B : \mathbb{B}^k \to \mathbb{B}$ and f a function $f : \mathbb{X}^k \to \mathbb{X}$. We say that f is a *monotonic extension* of f_B if $f(b) = f_B(b)$ for all $b \in \mathbb{B}^k$, and $b \sqsubseteq b'$ implies $f(b) \sqsubseteq f(b')$, for all $b, b' \in \mathbb{X}^k$. We say that f is *monotonic* if f is a monotonic extension of some f_B.

Lemma 1. *Let $f : \mathbb{X}^k \to \mathbb{X}$ be a monotonic extension of f_B, and $b_x \in \mathbb{X}^k$ such that $f(b_x) \in \mathbb{B}$. Then $f_B(b) = f(b_x)$ for all $b \in \mathbb{B}^k$ such that $b_x \sqsubseteq b$.*

Circuit Simulation and Undefined Values

Consider the simple pipelined adder circuit in Figs. 2 and 3. The design intent of the circuit is the computation of 8-bit or 16-bit addition in three pipestages B, C, and D. Data inputs are read from buses *datainB[1]* and *datainB[2]* in pipestage B, the actual computation takes place a cycle later in pipestage C, and the result is produced at the output bus *sumD* another cycle later in pipestage D. The input valid signal *avldB* is staged along with the data, and the input control signal *add8B* chooses between 8-bit and 16-bit addition. The expected behavior of the circuit *add* could be captured by properties such as *add8ok* in Fig. 4.

The most basic usage model of a traditional simulator in circuit validation is a single targeted test. The circuit *add* could be tested by generating stimulus and

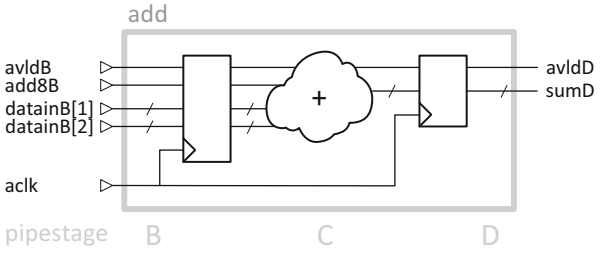


Fig. 2 Adder circuit schematic

```
‘define FF(q,i,clk) always_ff @(posedge clk) q <=i;     // flip-flop macro

module add( input aclk, avldB, add8B, [15:0] datainB[2:1],
            output reg avldD, reg [15:0] sumD );

  bit avldC, add8C; bit [15:0] datainC[2:1], mskC, datamskC[2:1], sumrawC, sumC;

  ‘FF( avldC, avldB, aclk ); ‘FF( add8C, add8B, aclk ); // flop pipestage B->C
  ‘FF( datainC, datainB, aclk );
  assign mskC        = {{8{~add8C}}, 8’hFF};            // build mask
  assign datamskC[1] = mskC & datainC[1];               // mask data
  assign datamskC[2] = mskC & datainC[2];
  assign sumrawC     = datamskC[1] + datamskC[2];       // add masked data
  assign sumC        = mskC & sumrawC;                  // mask sum
  ‘FF( avldD, avldC, aclk ); ‘FF( sumD, sumC, aclk );   // flop pipestage C->D
endmodule
```

Fig. 3 Adder circuit

```
property add8ok;   // correctness property for eight-bit addition
  @(posedge aclk)
    ( $past(avldB,2) && $past(add8B,2) ) |->
    ( sumD[7:0] == $past(datainB[1][7:0],2) + $past(datainB[2][7:0],2) )
endproperty
assert property(add8ok);
```

Fig. 4 Adder correctness property

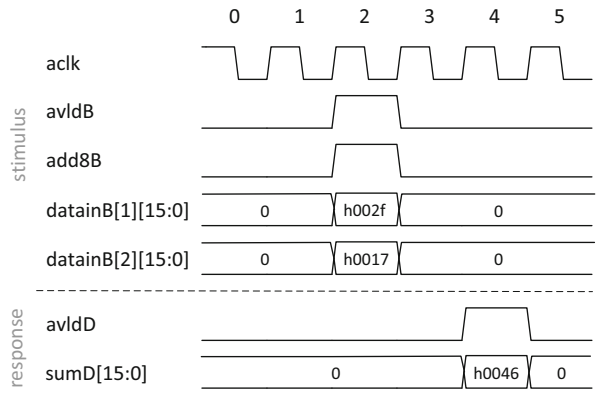


Fig. 5 Adder trace

simulating the circuit response as in Fig. 5. In this stimulus, a particular time, cycle 2, has been chosen to start an 8-bit addition operation in pipestage B; the control inputs *avldB* and *add8B* have been set accordingly; and specific input data values have been fixed. The simulator has also been started from a state where every state element is zero. This resulting trace could be attached to an external checker that samples *datainB* in cycle 2, computes the expected result, and compares it to *sumD* value in cycle 4. Alternatively, it could observed that the property *add8ok* is satisfied in cycle 4.

In this chapter, circuits are modelled as networks of three types of signals,

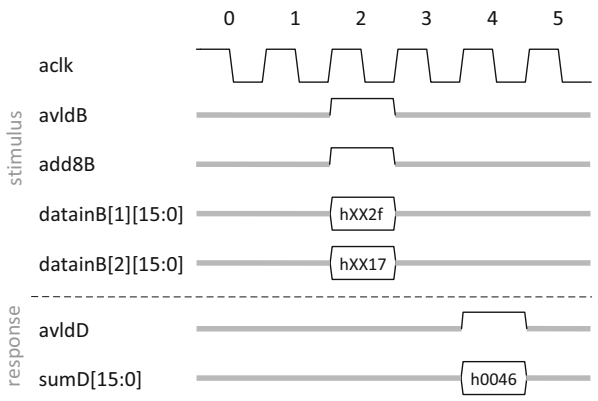
– *Inputs* receiving Boolean values from the outside
– *Combinational gates* computing Boolean functions
– *Delay elements* carrying values from one time moment to the next

We do not formalize the relation between the source code for a circuit and the mathematical model (in Definition 4 below), expecting the combinational gate functions, known as the *gate excitation functions*, to be self-evident monotonic extensions of common logical constructors, along the lines of Fig. 1. At the level of digital zero-delay modelling, all types of latches and flip-flops can be expressed in terms of combinational logic and delay signals, the basic building blocks for the model.

Circuits are assumed to be powered up in an unconstrained state and initialized by a reset sequence, typically by asserting a reset signal for some length of time. The instantaneous state of a circuit execution is modelled by an assignment of values to circuit signals (Definition 5) and the temporal dynamic behavior of a circuit by a finite or infinite sequence of states (Definition 6).

By applying the concept of monotonic extensions to the circuit gate functions, the set of values handled by simulation can be extended with the undefined value X, allowing stimulus that has Xs in addition to 0s and 1s. Note that an undefined value X in the stimulus does *not* mean that the user has not specified the value and the simulator picks either 0 or 1 randomly. It means that the signal is assigned the special value X, distinct from 0 and 1, and this value propagates through the gates of the circuit in the simulation according to rules like those in Fig. 1, resulting in internal and output values that may be either 0s, 1s, or Xs. Crucially, the monotonicity property of these rules then guarantees that every output signal that has a 0 or 1 value in the simulation has the same value in *every* simulation where the stimulus Xs are replaced with any combination of 0s and 1s. In this way, a single simulation trace with some undefined stimulus values X corresponds to a set of Boolean traces, and we gain knowledge about all these concrete traces with the cost of only a single simulation (Theorem 1).

For example, consider the stimulus and trace in Fig. 6, similar to Fig. 5, except that the stimulus associates X instead of 0 with signals and times other than the controls and the 8-bit input data for the main operation of interest. Assume also that this time, the simulator is started from a state where all state elements are Xs. From the waveform in Fig. 6, we can conclude that the circuit adds the two given 8-bit numbers correctly no matter what the preceding or following control or data signal values were, i.e., that there are no interferences between operations in successive cycles and that the result of the 8-bit addition operation is not affected by the high-input data bytes, a stronger statement than in the case of the Boolean simulation in Fig. 5.



Gray waveform bar denotes the undefined value X.

Fig. 6 Adder trace using undefined values

Mathematical Model of Circuit Simulation

Definitions 3, 4, 5, and 6 capture the circuit modelling intuition described above. The model determinism requirement in Definition 5 is used to exclude mathematically possible models that do not correspond to meaningful circuit designs. Determinism could be violated, for example, by inconsistent or isolated combinational loops, but such circuits are not considered well-formed designs in the first place. Theorem 1 then states the basic result that simulation traces computed using undefined values and monotonic extensions of the gate functions are abstractions of concrete circuit traces.

Definition 3. We write $\mathbb{N}$ for the set of natural numbers $\{0, 1, \ldots\}$ and $[n]$ for the set $\{i \in \mathbb{N} \mid i < n\}$ for every $n \in \mathbb{N}$. If S is a set and $n \in \mathbb{N}$, a *finite sequence* of length n over S is a function $s : [n] \to S$, and an *infinite sequence* over S is a function $s : \mathbb{N} \to S$. We denote the set of all finite sequences over S by $S^\star$. If $s \in S^\star$, the *length* of s, denoted $len(s)$, is the value n such that the domain of s is $[n]$. If $f : A \to B$ is a function, and $A' \subseteq A$, we write $f|_{A'}$ for the *restriction* of f to A', i.e., the function identical to f but restricted to domain A'.

Definition 4. A *circuit* C is a 7-tuple $C = (S_I, S_C, S_D, cfan, dfan, exc, reset)$ where:

– S_I, S_C, and S_D are finite nonoverlapping sets called the *input*, *combinational*, and *delay signals*, respectively. We define the set S of *signals* of C as $S = S_I \cup S_C \cup S_D$. The set S_D is also referred to as *state elements*.

- *cfan* is a function $cfan : S_C \to S^\star$, called the *combinational fanin* of C.
- *dfan* is a function $dfan : S_D \to S$, called the *delay fanin* of C.
- *exc* maps each $s \in S_C$ to a monotonic function $exc(s) : \mathbb{X}^{len(cfan(s))} \to \mathbb{X}$. We call $exc(s)$ the *excitation function* of the combinational signal s.
- *reset* is a finite non-empty sequence $reset \in (S_I \to \mathbb{X})^\star$, called the *reset sequence*. We define $rstlen = len(reset)$.

Definition 5. Let C be a circuit. We call a mapping $V_D : S_D \to \mathbb{X}$ a *state* of C, a mapping $V_I : S_I \to \mathbb{X}$ an *input state* of C, and a mapping $V : S \to \mathbb{X}$ a *full state* of C. We write ST, INP, and STF for the sets of all states, input states, and full states of the circuit C, respectively. The abstraction relation $\sqsubseteq$ extends to these sets as in Definition 1: $V \sqsubseteq V'$ if $V(s) \sqsubseteq V'(s)$ for all relevant s.

We define *UND*, the *maximally undefined state*, by $UND(s) = X$ for all $s \in S_D$.

We say that a full-state V is *combinationally consistent* if for every combinational signal $s \in S_C$, $V(s) = exc(s)(V(s_1), \ldots, V(s_k))$, where $cfan(s) = (s_1, \ldots, s_k)$. For full states V and V', we say that V' is *sequentially consistent* with V if $V'(s) = V(dfan(s))$ for every delay signal $s \in S_D$.

We say that a circuit is *deterministic* if for all states V_D and input states V_I, there is exactly one combinationally consistent full-state V such that $V|_{S_D} = V_D$ and $V|_{S_I} = V_I$.

We require all circuits to be deterministic.

Definition 6. Let C be a circuit. A *trace* of C is an infinite sequence of full states of C, i.e., an infinite sequence T over STF, such that for every $t \in \mathbb{N}$, $T(t)$ is combinationally consistent, and $T(t+1)$ is sequentially consistent with $T(t)$.

A *stimulus trace* of C is an infinite sequence of input states, i.e., an infinite sequence over INP. The notions of a *finite trace* and *finite stimulus trace* are defined analogously.

The abstraction relation $\sqsubseteq$ extends to the sets of (finite) traces and stimulus traces as in Definition 1: $T \sqsubseteq T'$ if $T(t)(s) \sqsubseteq T'(t)(s)$ for all t and s.

The *start state* of a (finite) trace T is defined as $start(T) = T(0)|_{S_D}$. The *end state* of a finite trace T with $len(T) > 0$ is defined as $end(T) : s \mapsto$ T(len(T)-1) ($dfan(s)$). The *stimulus* of a (finite) trace T is defined as $stim(T) : t \mapsto T(t)|_{S_I}$.

Due to the requirement that circuits are deterministic, for each state V_I and stimulus trace T_I, there is exactly one trace T such that $start(T) = V_I$ and $stim(T) = T_I$. We denote this trace by $trace(V_I, T_I)$. We write TR for the set of all traces of a circuit.

We call a stimulus trace T_I *initialized* if $reset \sqsubseteq T_I|_{[rstlen]}$. We call trace T *initialized* if $stim(T)$ is initialized.

We say that a trace or stimulus trace T is *Boolean* if $T(t)(s) \in \mathbb{B}$ for all $t \in \mathbb{N}$ and $s \in S$. We write TR_{INIT} for the set of all initialized Boolean traces of a circuit.

Theorem 1. *Let C be a circuit, V and V_X states, and T and T_X stimulus traces of C such that $V_X \sqsubseteq V$ and $T_X \sqsubseteq T$. Then $trace(V_X, T_X) \sqsubseteq trace(V, T)$.*

Proof. Follows from abstraction and monotonicity for deterministic circuits.

Circuit Properties

Conceptually, circuit properties are predicates that for every trace and every point of a trace are either true or false at that point. They can be modelled by functions that map traces to sequences of truth values. A multitude of formalisms exists for describing circuit properties, for example different temporal logics. In the examples of this chapter, properties are written as SystemVerilog Assertions (SVA) (IEEE standard for SystemVerilog–unified hardware design 2018). As with circuits, we do not formalize the relation between the SVA source code and the mathematical property, expecting the relation to be self-evident for the common constructors.

Symbolic simulation targets a simple yet very useful set of properties, fixed time-window invariants, defined as the minimal set that contains all direct signal references and is closed under fixed time offsets and Boolean operators (Definition 7). In traditional linear temporal logic terms, this set coincides with the set of formulas built from atomic propositions, Boolean operators, and "next-time" and "previous-time" temporal operators, wrapped in a single "always" operator. The example SVA properties in this chapter use only direct signal references, Boolean operators, the "$past" temporal operator, and the overlapping implication operator |–> without a time delay.

When determining the validity of a property over a circuit, only Boolean traces are considered, reflecting the intuition that the underlying reality that is modelled is Boolean and that the use of the undefined value X is a modelling artifact. The initialization phase of traces is also explicitly ignored: Only states after initialization count for the validity of a property (Definition 7).

Simulation using the undefined value X allows us to validate a universal invariant property based on a single instance of it. In other words, if a property holds at one fixed time with a fixed stimulus trace starting from a maximally undefined state, then the same property holds in *every time point of every trace* of the circuit, as long as the trace agrees with the Boolean 0 and 1 values present in the stimulus (Theorem 2). Note that it is essential to the argument that the simulation trace validating a property starts from an unconstrained state. The reset sequence and circuit initialization play no role in the consideration. The downside of this aspect is that the property is analyzed over a wider set of behaviors than necessary, including traces that are not properly initialized.

Returning to the example circuit *add* in Fig. 3, by iterating over the 2^{16} possible input data values and repeating the simulation in Fig. 6, complete correctness of the circuit for the 8-bit addition operation could be established, validating the property *add8ok* of Fig. 4 at all points of all possible traces of the circuit.

Mathematical Model of Circuit Properties

Definition 7 introduces the concept of a *property expression*, capturing the class of properties targeted by symbolic simulation as discussed above, and Lemma 2 states that X-abstraction is validity-preserving in this class of properties. To aid theory development, it is useful to be able to convert stimulus contents into equivalent properties. Definition 8 shows how to do this: stimulus restricts inputs only in those points where it assigns a 0 or 1 value, and these Booleans can be captured by a property. Lemma 3 relates such properties and X-abstraction. Theorem 2 then states the fundamental result that the validity of a property over all times and all traces of a circuit can be established by looking at a single property instance over a single simulation trace with X values.

Definition 7. A *property* is an expression with the syntax $P ::= s \mid P@t \mid f(P_1, \ldots, P_k)$, where $s \in S$ is a signal of circuit C; $P, P_1, \ldots, P_k$ are properties; the *time offset* t is an integer; and $f : \mathbb{X}^k \to \mathbb{X}$ is any monotonic function of arity k.

We define function $valid(P) : TR \to (\mathbb{N} \to \mathbb{X})$ for all traces $T \in TR$ and $t \in \mathbb{N}$ by:

- $valid(s)(T)(t) = T(t)(s)$, where $s \in S$ is a signal of C
- $valid(P@t')(T)(t) = valid(P)(T)(t+t')$ when $t+t' \geq 0$, and X when t+t'< 0
- $valid(f(P_1, \ldots, P_k))(T)(t) = f(valid(P_1)(T)(t), \ldots, valid(P_k)(T)(t))$

We say that P is *valid at time t of trace T* and write $T, t \models P$ if valid(P)(T)(t)=1.

We say that a property P is *valid over circuit C*, and write $C \models P$ if $T, t \models P$ for all initialized Boolean traces $T \in TR_{INIT}$ and all times $t \geq rstlen$.

Lemma 2. *Let C be a circuit, P a property; V, V_X states; T, T_X stimulus traces of C; and $t \in \mathbb{N}$ such that $V_X \sqsubseteq V$, $T_X \sqsubseteq T$ and $trace(V_X, T_X), t \models P$. Then $trace(V, T), t \models P$.*

Proof. From Theorem 1 and the monotonicity of the property functions.

Definition 8. Let T, T' be (stimulus) traces. We define $zeros(T), ones(T) \subseteq \mathbb{N} \times S$ by $zeros(T) = \{(t, s) \mid T(t)(s) = 0\}$ and $ones(T) = \{(t, s) \mid T(t)(s) = 1\}$.

We say that T *contains Booleans of* T' and write $T' \sqsubseteq_{\mathbb{B}} T$ if $ones(T') \subseteq ones(T)$ and $zeros(T') \subseteq zeros(T)$.

We define $char(T) = \bigwedge\{\neg s@t \mid (t, s) \in zeros(T)\} \wedge \bigwedge\{s@t \mid (t, s) \in ones(T)\}$ and call it the *characteristic property* of T.

Lemma 3. *Let T be a stimulus trace and T' a trace. Then $T', 0 \models char(T)$ if and only if $T \sqsubseteq_{\mathbb{B}} stim(T')$ if and only if $T \sqsubseteq stim(T')$.*

Theorem 2. *Let C be a circuit, P a property, and T a stimulus trace of C, and $t \in \mathbb{N}$, such that $trace(UND, T), t \models P$. Then $C \models char(T) \Rightarrow P@t$.*

Proof. Take any Boolean trace T' and $t' \geq rstlen$ such that $T', t' \models char(T)$. Let T'' be the suffix of T' starting from point t'. Then $T'', 0 \models char(T)$, implying $T \sqsubseteq stim(T'')$ by Lemma 3. As $UND \sqsubseteq start(T'')$, then $T'', t \models P$ by Lemma 2, hence $T', (t' + t) \models P$ and $T', t' \models P@t$.

Symbolic Simulation

Symbolic Computation

In *symbolic computation*, the set of Boolean values is further extended from 0, 1, and X by a collection of *symbolic variables*. These symbolic variables are effectively names of values, representing sets of possible concrete values. With symbolic computation, it is possible to calculate the value of an expression for a set of argument values in one evaluation.

Conceptually, a symbolic Boolean expression is an object that refers to zero or more symbolic variables And, given an assignment of Boolean values to those variables, will return a Boolean value. Mathematically, an assignment of Boolean values to symbolic variables corresponds to a function from the set of symbolic variables to Booleans, and a symbolic Boolean to a function from such assignments to Booleans, as in Definition 9. The process of instantiating a symbolic Boolean with a given variable assignment to yield an ordinary Boolean corresponds to function application. Each symbolic variable can be naturally viewed as a symbolic Boolean, and every function on Booleans induces an analogous function on symbolic Booleans, a process called the *symbolic lift*.

Definition 9. Let V be a finite set, the elements of which we call *symbolic variables*. We define the set of *variable assignments VA* as the set $(V \to \mathbb{B})$, and the set of *symbolic booleans* $\mathbb{S}$ as the set $(VA \to \mathbb{X})$.

For every variable assignment $a \in VA$, we define the *symbolic instantiation by* a, $inst(a) \in (\mathbb{S} \to \mathbb{X})$ as the function $inst(a) : s \mapsto s(a)$. We extend the symbolic instantiation $inst(a)$ to any function $f : A_1 \to \ldots (A_n \to \mathbb{S})$, for any sets $A_1, \ldots, A_n$, by defining $inst(a)(f)(a_1) \ldots (a_n) = inst(a)(f(a_1) \ldots (a_n))$.

The *symbolic lift* of a function $f : \mathbb{X}^k \to \mathbb{X}$ is the function $symb(f) : \mathbb{S}^k \to \mathbb{S}$ defined by $symb(f)(s_1, \ldots, s_k) : a \mapsto f(s_1(a), \ldots, s_k(a))$. For every symbolic variable $v \in V$, we define $symb(v) \in \mathbb{S}$ as the function $symb(v) : a \mapsto a(v)$. Where it causes no confusion, we write 0, 1, X, and v for the functions $symb(0)$, $symb(1)$, $symb(X)$ and $symb(v)$, for any symbolic variable $v \in V$.

The abstraction relation $\sqsubseteq$ extends to $s, s' \in \mathbb{S}$ as in Definition 1: $s \sqsubseteq s'$ if $s(a) \sqsubseteq s'(a)$ for all $a \in VA$.

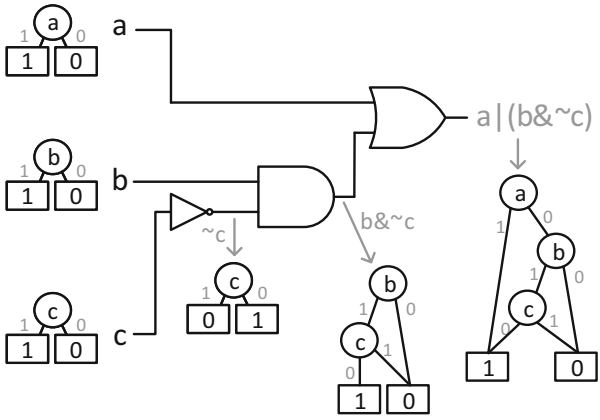


Fig. 7 Symbolic computation using a BDD representation

There are several different techniques for representing symbolic Boolean expressions as graph data structures. The key aspects of any such technique are the size of the representation and the efficiency of computing the lifted Boolean operations directly on the representation.

The most traditional approach uses binary decision diagrams (BDDs) (Bryant 1986). Figure 7 contains an example of a sequence of symbolic computations using BDDs. The great advantage of BDDs is that they provide a *canonical* representation, i.e., each possible Boolean function over a set of variables has one unique representation. In the downside, BDDs require a global ordering of variable names, and the size of the representation can vary dramatically depending on the variable ordering applied. Certain logic functions also simply do not have concise representations as BDDs irrespective of the ordering, for example, arithmetic multiplication (Bryant 1986).

As an alternative to BDDs, symbolic expressions are also frequently represented using and-inverter graphs (AIGs) (Kuehlmann et al. 2002) or some similar data structures (Bjesse and Boralv 2004) that typically allow a more concise representation at the expense of canonicity.

As a slight departure from tradition, to help theory development, we have chosen the image set of symbolic Booleans in Definition 9 to be $\mathbb{X}$, including the undefined value X, instead of the more common $\mathbb{B} = \{0, 1\}$. The symbolic Booleans in the sense of Definition 9 can easily be implemented in terms of standard symbolic Booleans, for example, with a dual-rail representation (Bryant and Seger 1990).

Simulation with Symbolic Values

In a bit-level symbolic simulator, each symbolic variable corresponds to the set of Boolean values 0 and 1. Note here that symbolic variables are elements from a domain totally distinct from that of circuit signals. In symbolic simulation, symbolic

variables are associated with specific signals at specific times in the stimulus. Associating a variable with a signal at a time effectively says that the value of the signal at the time is either 0 or 1 (and not X) and that the actual value is not fixed, but instead the symbolic variable is used to refer to what the value is.

Consider the symbolic stimulus and trace in Fig. 8 for the adder circuit of Fig. 3. In addition to the 0, 1, and X values, the stimulus contains the 16 symbolic variables $a[7] \ldots a[0]$ and $b[7] \ldots b[0]$. In the simulation, the symbolic values propagate alongside the 0, 1, and X values, and in each logic gate, they are combined with each other to result in either a logical expression on the symbolic variables or a 0, 1, or X value.

For example, the symbolic variable $a[0]$ is associated with signal *datainB[1][0]* in simulation cycle 2, $b[0]$ with *datainB[2][0]* and so on. Variable $a[0]$ propagates from signal *datainB[1][0]* in cycle 2 to *datainC[1][0]* in cycle 3 through a flip-flop, then through *datamskC[1][0]* and to *sumrawC[0]*, where it is combined with $b[0]$ to yield $(a[0]\&!b[0]+!a[0]\&b[0])$. This value propagates to *sumC[0]* in cycle 3 and finally to output *sumD[0]* in cycle 4 through a flip-flop. Figure 9 depicts this path, annotated with the signal values in the cycle relevant to each signal. In all other simulation cycles, all signals in the picture have the value X, except for the constant *mskC[0]*.

The symbolic expressions for the result bus *sumD* in cycle 4 can be extracted from the trace and compared to a reference model result computed by adding together the two symbolic variable vectors $\{a[7] \ldots a[0]\}$ and $\{b[7] \ldots b[0]\}$. Alternatively, the SVA correctness property *add8ok* of Fig. 4 can be included in the

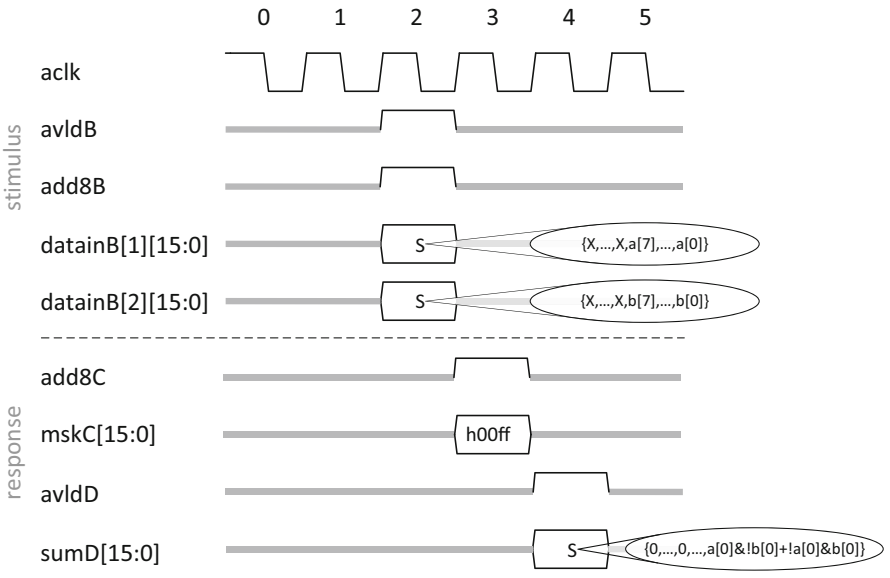


Fig. 8 Adder trace using symbolic values

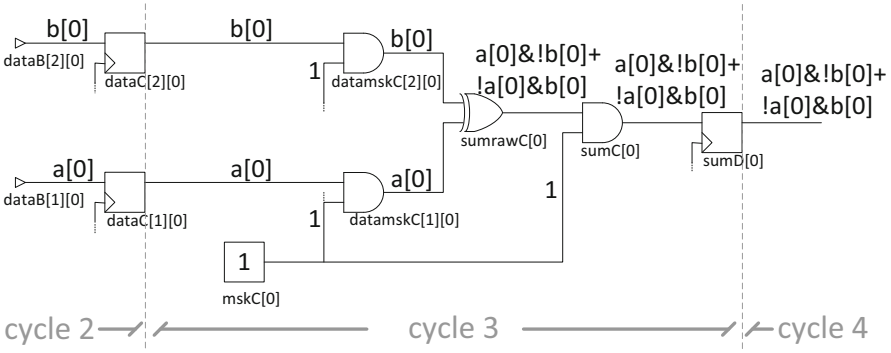


Fig. 9 Simulation using symbolic values

symbolic simulation in Fig. 8, performing the addition of the input data values and comparison to the result bus as a part of the property simulation in cycle 4.

The idea of symbolic simulation as a verification method starts from the observation that the stimulus in Fig. 8 does not place any restrictions on inputs besides the Boolean values on the control signals *avldB* and *add8B* in cycle 2. The symbolic variables associated with the data bus *datainB* in cycle 2 of the stimulus allow every possible value combination to occur, and in the other cycles, both control and data inputs have the undefined value X. The start state of the simulation is also unrestricted. In this way, the single symbolic trace represents every point of every Boolean trace of the circuit that agrees on the fixed Boolean values on the two control signals. Since the property *add8ok* holds in the symbolic trace, we conclude that it holds at every point of every Boolean trace of the system, for all possible input patterns. Just for the data, the symbolic simulation is doing the work of 2^{16} traditional simulations, one for each possible assignment of 0/1 values to the 16 symbolic variables. This intuition is captured mathematically in Theorem 3 below.

Both Xs and symbolic variables in stimulus represent lack of information in that they do not uniquely describe a value. However, their nature is different. The use of X is an abstraction mechanism, while the use of symbolic variables is merely a vehicle for doing the work of multiple concrete simulations in one simulation. A key difference is illustrated in Fig. 10: Xs are pessimistic whereas symbolic values are not. Also, there is only one X, whereas each symbolic variable refers to a specific distinct Boolean value, although not to a fixed value. If an X is associated with two different signals, there is no relation between the values of the signals. If the same symbolic variable a is associated with two different signals, they are implicitly restricted to have the same Boolean value, either both 0 or both 1.

In practical symbolic simulation, the size of the symbolic expressions flowing in the wires during the simulation is the most crucial complexity metric and almost always the limiting factor determining the applicability of the method, as simulation with 0s and 1s is cheap and simulation with Xs is even cheaper. Later in this chapter, we look at a host of techniques for managing this complexity.

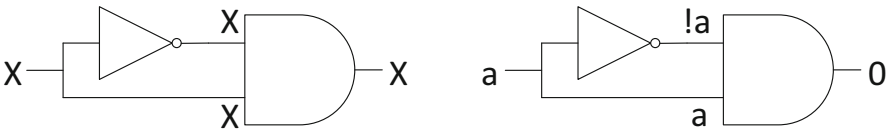


Fig. 10 Undefined value vs symbolic variable propagation

When symbolic values are expressed using BDDs, the comparison between circuit and reference model results is trivial. For the two to agree, the expressions computed by both must be identical because of canonicity.

When AIGs are used to represent symbolic values, the simulation needs to be connected to an external SAT solver to determine the consistency between the circuit and a reference model or the validity of a checker property. One way of looking at the usage of AIGs instead of BDDs for the simulation is as a tradeoff of complexity during simulation versus complexity after the simulation.

Mathematical Model of Symbolic Simulation

To mathematically develop the intuition above, Definition 10 extends the basic notions to the symbolic domain, and Lemma 4 establishes soundness of symbolic simulation. The intuition that symbolic stimulus does not restrict a signal when it assigns an undefined or symbolic value to it is not entirely accurate though. For example, a stimulus that assigns the same symbolic variable to two different signals only covers behaviors where the signals have the same value. To characterize the cases where the intuition holds, we formulate the side condition of *universality* in Definition 11.

Theorem 3 states the fundamental result justifying symbolic simulation as a verification method: Validity of a property over a single universal symbolic stimulus trace starting from an unrestricted state validates the property for all traces and times. Definition 12 and Corollary 1 then articulate a common usage model with a more straightforward side condition, requiring each symbolic variable to be used at most once in the stimulus. All examples in this chapter fall under Corollary 1. Certain other usage models, such as symbolic indexing (Adams et al. 2007), require more general forms of stimulus.

Definition 10. We define the concepts of *symbolic state* and *symbolic trace*; the functions $valid$, $ones$, $zeros$, and $char$; the relations $\sqsubseteq$, $\sqsubseteq_{\mathbb{B}}$; and related concepts for symbolic traces as in Definitions 5, 6, 7 and 8, by replacing the set $\mathbb{X}$ with $\mathbb{S}$, and by replacing all references to circuit excitation and property functions and 0, 1, and X with their symbolic lifts. Every (stimulus) trace T can be viewed as a symbolic (stimulus) trace by replacing every 0, 1, and X in the image of T with their symbolic lifts.

Lemma 4. *Let V_I be a symbolic state and T_I a symbolic stimulus trace of a circuit C, let $T = trace(V_I, T_I)$, and let $a \in VA$ be a variable assignment. Then, $inst(a)(T)$ is a trace of C and $inst(a)(T) = trace(inst(a)(V_I), inst(a)(T_I))$.*

Further, let P be a property and $t \in \mathbb{N}$. If $T, t \models P$, then $inst(a)(T), t \models P$

Lemma 5. *Let T be a symbolic stimulus trace and T' a trace. Then $T', 0 \models char(T)$ if and only if $T \sqsubseteq_{\mathbb{B}} stim(T')$.*

Definition 11. We say that a symbolic stimulus trace T is *universal* if for every Boolean stimulus trace T' such that $T \sqsubseteq_{\mathbb{B}} T'$, there exists $a \in VA$ such that $inst(a)(T) \sqsubseteq T'$.

Theorem 3. *Let C be a circuit, P a property, and T a universal symbolic stimulus trace of C, and $t \in \mathbb{N}$, such that $trace(UND, T), t \models P$. Then $C \models char(T) \Rightarrow P@t$.*

Proof. *Take any Boolean trace T' and $t' \geq rstlen$ such that $T', t' \models char(T)$. Let T'' be the suffix of T' starting from point t'. Then $T'', 0 \models char(T)$, implying $T \sqsubseteq_{\mathbb{B}} stim(T'')$ by Lemma 5. As T is universal, $inst(a)(T) \sqsubseteq stim(T'')$ for some $a \in VA$. Since $trace(UND, T), t \models P$, by Lemma 4 also $trace(UND, inst(a)(T)), t \models P$. As $UND \sqsubseteq start(T'')$, Lemma 2 then implies $T'', t \models P$, hence $T', (t' + t) \models P$ and $T', t' \models P@t$.*

Definition 12. We say that a symbolic stimulus trace T is *simple* if

- $T(t)(s)$ is either 0, 1, X or $symb(v)$ for some $v \in V$, for all $(t, s) \in \mathbb{N} \times S_I$, and
- for every $v \in V$, there is at most one $(t, s) \in \mathbb{N} \times S_I$ such that $T(t)(s) = symb(v)$.

Lemma 6. *Every simple symbolic stimulus trace T is universal.*

Corollary 1. *Let C be a circuit, P a property, and T a simple symbolic stimulus trace of C, and $t \in \mathbb{N}$, such that $trace(UND, T), t \models P$.*

Then $C \models char(T) \Rightarrow P@t$.

Practical Considerations

Algorithmically the verification of a property with symbolic simulation is quite different from a traditional model checker that would start from the set of initial states of the system, then compute a representation of the reachable states as a fixed point of the transition relation, and verify along the way that the expected property holds for all reachable states. First, in symbolic simulation, there is no notion of an initial state or reachable state. Instead, the universality of the verified property relies on the unrestricted start state of the symbolic simulation trace. Second, no

fixed-point computation takes place in the symbolic simulation, and the property is verified relative to one single fixed time point of the symbolic trace. As the fixed-point computation is the expensive part of model checking, this latter aspect sets the two methods far apart regarding computational complexity. Of course, it also restricts what can be verified.

The practice of formal verification with symbolic simulation is also very different from common model checking. The former is at heart an interactive computer-aided verification method guided by human intuition about the circuit under verification. The latter strives to be a fully automated approach that still in reality requires careful human intervention in the ways the tool is invoked to guide it past complexity bottlenecks.

While a successful verification of a property with symbolic simulation always guarantees its validity, the inverse is not true. The validation of a property often fails just because some signal values it refers to are Xs in the simulation: Without knowing what the values are, the validity of the property cannot be determined. A large part of the human verification effort goes to root-causing these Xs and strengthening the stimulus by associating symbolic variables with more signals.

From user perspective, verification of a property by focusing on a single instance of it in symbolic simulation is somewhat analogous to looking at a single instance of a property in bounded model checking (BMC). In both cases, there is a fixed-length sequence from a start state, on which the property either holds or is violated. One difference is that in BMC, the start state is initialized, whereas in symbolic simulation, it is unrestricted. Also, in BMC, the sequence is represented by a series of unrolled next-state relations, whereas in symbolic simulation, the effects of the circuit computation are gradually accumulated into the expressions flowing in the wires in the simulation. Considering the unrestricted start state, symbolic simulation is conceptually similar to an extreme case of k-induction, for the case $k = 0$. However, the usage models of the two methods are quite disparate, one stressing automation and the other user guidance, and it is unclear what practical implications could be drawn from parallels between them.

Simulation Scope Control

Property Triggers

When using symbolic simulation as a formal verification method, it is advantageous to consider the verification target properties as being written in the form:

$$(trig_1 \wedge \ldots \wedge trig_n) \Rightarrow (goal_1 \wedge \ldots \wedge goal_m)$$

where the individual triggers $trig_i$ and verification goals $goal_i$ are fixed time window properties. Of course, every fixed time window invariant can be trivially expressed in this form by leaving the trigger empty and having the given property as the single goal. However, the efficiency of the method is highly dependent on

the triggering conditions restricting both the scope of circuit logic that needs to be simulated and the size of the expressions in the simulation. If symbolic variables are associated with all inputs in the stimulus without any triggering conditions, all the simulation does is to copy the circuit logic syntactically into the expressions flowing in the wires.

Consider the 8-bit adder example of Fig. 3 and the propagation of values in the pipeline with the stimulus in Fig. 8. Corresponding to the triggers *avldB* and *add8B* of the property *add8ok* in Fig. 4, the stimulus sets the values of these signals to 1 at cycle 2. In the simulation, due to the values propagating in the pipeline, the signal *add8C* has value 1 in cycle 3. This value 1 is used to compute the mask vector *mskC* leading to the high bytes of *datainC* being zeroed in *datamskC* before the addition takes place in signal *sumrawC*. Now, when adder logic is simulated, only 8-bit symbolic addition needs to be computed, even though the circuit contains logic for 16-bit addition.

The discussion above glossed over the detail of *how* the triggers of the property *add8ok* in Fig. 4 became the fixed values for *avldB* and *add8B* in cycle 2 in the stimulus of Fig. 8. It is not difficult to imagine an algorithm, though, that would derive such fixed values from the triggers of a given property and a user-supplied reference cycle for checking its goals, in this instance the property *add8ok* and cycle 4. The question is more general, however. In the *add8ok* example, the triggers are particularly simple and directly correspond to fixed Boolean stimulus values. This is generally not the case, and individual trigger conditions may be satisfied by multiple alternative value assignments. Yet, it is important to restrict the scope of the simulation only to the cases where the triggers are satisfied. Any other cases do not matter for the verification goals, and incurring the cost of simulating such cases is effort wasted.

To illustrate this issue, let us introduce another execution unit *mul* for multiplication and combine the adder and multiplier to a simple ALU with different latencies, two cycles for *add* and three cycles for *mul*, as in Figs. 11, 12, and 13. The ALU circuit contains powerup logic for the subunits, decoding the operation code *opA*, and optional power-saving logic that turns on the clock to a subunit only when there is an operation going through it. Each subunit powers up for five cycles when needed. A reset signal *rst* turns on the clocks and clears the valid signals going through the units. A high power mode switch *PWRHI* overrides the power-saving logic and forces clocks to both subunits to toggle freely. For the moment being, assume the *PWRHI* override to be 1.

The 8-bit adder correctness property can be moved to the ALU level as in Fig. 14. In the context of the ALU circuit, however, it is not sufficient to assert just the valid signal and the operation control for the ADD8 operation, as the presence of the three-cycle MUL creates a pipeline hazard. For the addition operation to work, there cannot be a MUL operation starting a cycle earlier. This requirement can be incorporated into the correctness property as a third trigger as in Fig. 15.

Consider then the verification of the property *alu8okB* of Fig. 15, with focus on cycle 4 for checking the goal of the property. Informally this could be done by

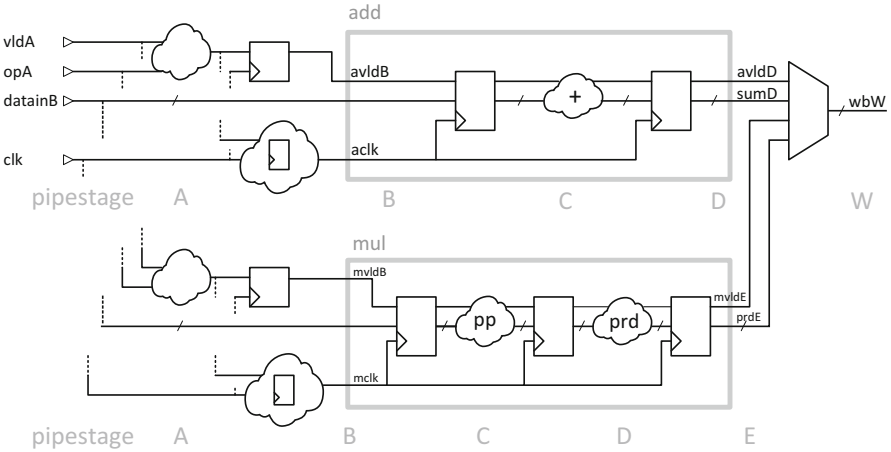


Fig. 11 ALU circuit schematic

```
module mul( input mclk, mvldB, [15:0] datainB [2:1],
            output reg mvldE, reg [15:0] prdE );
  bit mvldC, mvldD; bit [15:0] datainC [2:1], ppC [15:0], ppD [15:0], prdD;
  integer i;

  'FF( mvldC, mvldB, mclk ); 'FF( datainC, datainB, mclk ); // flop pipestage B->C
  always_comb for (i=0;i<16;i=i+1)                          // partial products
     ppC[i]=(datainC[1]<<i)&{16{datainC[2][i]}};
  'FF( mvldD, mvldC, mclk ); 'FF( ppD, ppC, mclk );         // flop pipestage C->D
  always_comb begin
     prdD=16'b0;for (i=0;i<16;i=i+1) prdD=prdD+ppD[i];end   // sum partial products
  'FF( mvldE, mvldD, mclk ); 'FF( prdE, prdD, mclk );       // flop pipestage D->E
endmodule
```

Fig. 12 Multiplier circuit

simulating an ADD8 operation that starts in pipestage A in cycle 1 and writes back the result in cycle 4. The first and second triggers of *alu8okB* correspond naturally to fixed values $vldA = 1$ and $opA = ADD8 = 01$ in cycle 1. The third trigger can be satisfied in two ways: either when $vldA \neq 1$ or when $opA \neq MUL = 11$ in cycle 0. It can be seen informally from the circuit logic in Fig. 16 that under this restriction the internal signal *mvldA* is equal to 0 in cycle 0, and the value propagates to *mvldE* in cycle 4, allowing the ADD8 result through the write-back mux and ignoring any value coming from the multiplier. However, there is no fixed value assignment that would match the third trigger. *Some* assignment could be picked, but that would ignore the other ways of satisfying the trigger. Iterating over all possible assignments that make the trigger true could be considered, but in general, the number of such assignments is exponential in the number of bits involved. An optimal solution would be to symbolically simulate *all* cases where the trigger is satisfied, and *only* those, in a *single* simulation. The next subsection focuses on techniques enabling precisely that.

```
parameter NOP = 2'b00, ADD8 = 2'b01, ADD16 = 2'b10, MUL = 2'b11, PWRHI = 1'b1;

module alu( input rst, clk, vldA, [1:0] opA, [15:0] datainB [2:1],
            output reg vldW, reg [15:0] wbW );
  bit aclk, avldA, add8A, avldB, add8B, avldD, mclk, mvldA, mvldB, mvldE;
  bit [15:0] sumD, prdE; bit [4:0] avldN, mvldN;

  assign avldA = vldA & ((opA==ADD8)|(opA==ADD16))&~rst;// add powerup
  assign mvldA = vldA & (opA==MUL) & ~rst;              // mul powerup
  assign add8A = (opA==ADD8);                           // add datasize
  'FF( {avldB,add8B,mvldB}, {avldA,add8A,mvldA}, clk ); // flop pipestage A->B

  'FF( avldN, { avldN, (avldA|rst) }, clk );            // shift registers for
  'FF( mvldN, { mvldN, (mvldA|rst) }, clk );            // add/mul clock enables
  assign aclk = clk & ((|avldN) | PWRHI );              // add clock
  assign mclk = clk & ((|mvldN) | PWRHI );              // mul clock

  add add( aclk, avldB, add8B, datainB, avldD, sumD );  // add, pipestages B->C->D
  mul mul( mclk, mvldB,        datainB, mvldE, prdE );  // mul, pipestages B->C->D->E

  assign vldW  = avldD | mvldE ;                        // result valid, D/E==W
  always_comb unique case (1'b1)
    avldD: wbW = sumD; mvldE: wbW = prdE; endcase;      // result mux,   D/E==W
endmodule
```

Fig. 13 ALU circuit

```
property alu8ok; // correctness property for eight-bit addition at ALU level
  @(posedge clk)
    ( $past(vldA,3) &&
      ( $past(opA,3)==ADD8 )
    ) |->
    ( wbW[7:0] == $past(datainB[1][7:0],2) + $past(datainB[2][7:0],2) )
endproperty
```

Fig. 14 Adder correctness property at ALU level

```
property alu8okB;
  @(posedge clk)
    ( $past(vldA,3) &&
      ( $past(opA,3)==ADD8 ) &&
      ~( $past(vldA,4) && ($past(opA,4)==MUL) )
    ) |->
    ( wbW[7:0] == $past(datainB[1][7:0],2) + $past(datainB[2][7:0],2) )
endproperty
```

Fig. 15 Adder correctness property with scheduling restriction

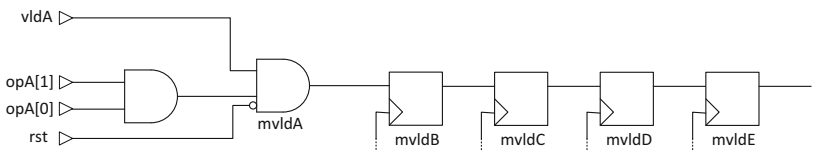


Fig. 16 Multiplier valid signal pipeline

Scope Reduction by Triggers

In summary, to contain symbolic simulation complexity, it is essential that property triggers are used to reduce the scope of simulated circuit logic, even when those triggers do not directly map to fixed stimulus values. Depending on whether the symbolic representations used in the simulation are based on BDDs or AIGs, there are two different algorithmic techniques to achieve this goal. For BDD-based simulation, the standard approach is the method of *parametric substitutions* (Jones 2002; Aagaard et al. 1999a). For AIGs, calls to a SAT-based simplifier can be embedded to the simulator. We do not discuss either of these methods in detail, focusing instead on the rationale and their effects.

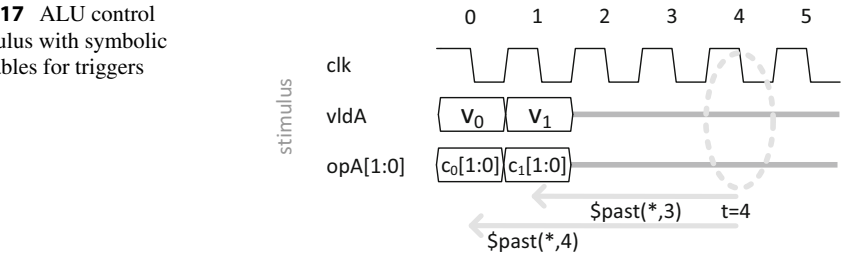
The basic setup for the parametric substitution algorithm is the verification of an implication $C(v_1, \ldots, v_n) \Rightarrow D(v_1, \ldots, v_n)$ between two symbolic Boolean expressions C and D over a set of symbolic variables $v_1, \ldots, v_n$, where the assumption C restricts D in a fashion that makes the symbolic evaluation of D less costly. The algorithm creates a mapping $v_i \mapsto p_i$ from each v_i to a symbolic expression p_i such that when the symbolic variables in p_i's range over all possible values, the values of the symbolic vector $p_1, \ldots, p_n$ range exactly over the set of assignments to $v_1, \ldots, v_n$ for which the condition C is true. Then, the implication can be verified by checking whether $D(p_1, \ldots, p_n)$ is true, evaluating D only for argument values for which C holds. For in-depth technical discussion of parametric substitutions, see Jones (2002) and Aagaard et al. (1999a), and for a related advanced technique called *universal Boolean functional vectors*, see Bingham (2015).

In the symbolic simulation context, the aim is to verify the goal conditions for exactly the set of scenarios for which the trigger conditions hold. With the parametric substitution algorithm, this is done in three steps:

1. Simulate the triggers on stimulus which associates symbolic variables with all signals and times the triggers refer to.
2. Compute parametric substitution from the trigger expressions from step 1.
3. Apply the substitution from step 2 to the stimulus and simulate the property goals.

Returning to property *alu8okB* in Fig. 15 with focus on cycle 4, the triggers refer to signals *vldA* and *opA* in cycles 0 and 1, and step 1 can use the stimulus in Fig. 17. The expression for the first trigger *$past(vldA,3)* on this stimulus is v_1. For the second trigger, *$past(opA,3)==ADD8* it is $c_1[1] \wedge \neg c_1[0]$. For these two triggers, step 2 returns the substitution $v_1 \mapsto 1$, $c_1[1] \mapsto 1$, $c_1[0] \mapsto 0$, the only satisfying assignment. For the third trigger, the process from trigger to expression to substitution is:

$$\text{~(\$past(vldA,4)\&\& (\$past(opA,4)==MUL))} \longrightarrow \neg(v_0 \wedge c_0[1] \wedge c_0[0]) \longrightarrow \begin{array}{l} c_0[1] \mapsto c_0[1] \\ c_0[0] \mapsto c_0[0] \\ v_0 \mapsto (v_0 \wedge (\neg c_0[1] \vee \neg c_0[0])) \end{array}$$



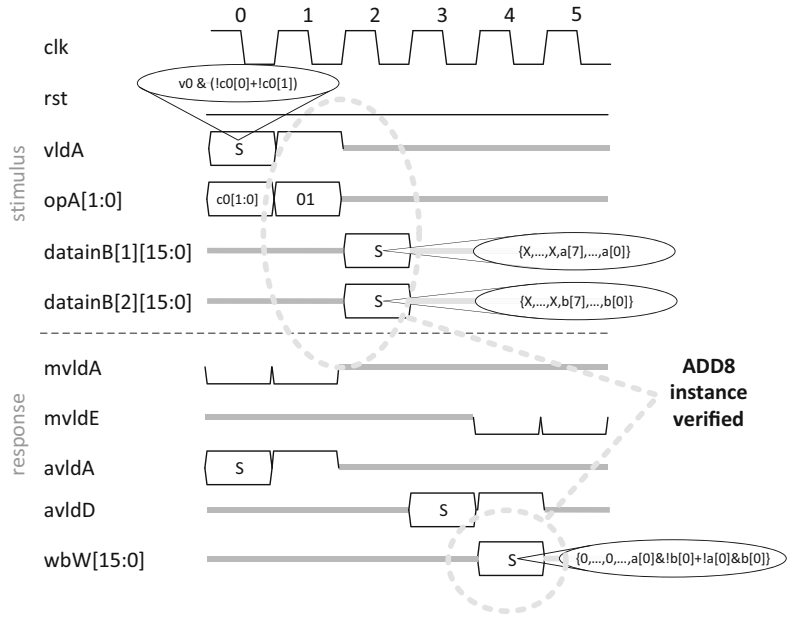


Fig. 17 ALU control stimulus with symbolic variables for triggers

Fig. 18 ALU waveform with parametric substitution

The three target expressions of the substitutions refer to the three variables v_0, $c_0[1]$, and $c_0[1]$. The values of these three expressions for the eight possible Boolean assignments for the variables cover every combination except 1-1-1, i.e., the exact set of assignments that makes $\neg(v_0 \wedge c_0[1] \wedge c_0[0])$ true.

At step 3 in the parametric substitution process, the substitutions are applied to the stimulus of Fig. 17, resulting in the stimulus in Fig. 18. Independent symbolic variables are added for the data bus, and finally the goal of the property *alu8okB* of Fig. 15 is simulated. The resulting simulation covers now exactly the cases where the trigger conditions hold. With the stimulus of Fig. 18, the computation of *mvldA* in cycle 0 occurs as in Fig. 19, resulting in the value 0. When parametric substitutions are used in a BDD-based simulation, such internal simplifications happen automatically, due to the canonicity of BDDs.

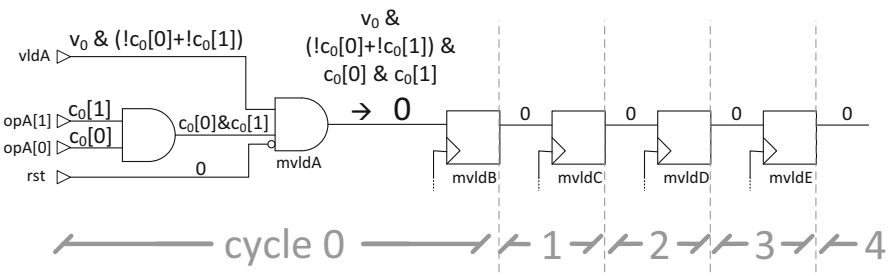


Fig. 19 Internal simplification

If AIGs are used for logical expressions instead of BDDs, it is still advantageous to derive all the constant signal assignments from the triggers, as they help to simplify circuit logic. The rest of the parametric substitution method, on the other hand, is not useful with AIGs. Instead, internal simplifications need to be explicitly enabled in the simulator, for example, by speculative SAT calls checking whether the expression for an internal signal in the simulation is equivalent to a Boolean 0 or 1 under the trigger conditions. The limiting factor in this respect is the number of circuit signals in the simulation, which means that the time that can be spent on simplification per signal is very short, or the user must be able to judiciously guide the tool to attempt simplification only on certain signals.

The usual default strategy is to attempt to maximize internal simplifications because of the resulting reduction in simulation scope. This also helps goal checking: if the simulation is done using BDDs and all triggers are used for parametric substitutions, it is sufficient to just check that all goals evaluate to 1s in the simulation. However, there are cases when the cost of computing the simplifications, especially the cost of computing the parametric substitutions, exceeds the savings it has provided. In these cases, it may be better not to use some of the triggers for simplification. Instead, they need to be factored in when checking the validity of the verification goals after the simulation. If the simulation is done using non-canonical symbolic representations with a SAT call in the end, all triggers are be used as assumptions for the goal checking already by default. When simulation is done using BDDs, however, extra post-simulation work is needed to check whether the symbolic expression for the triggers imply the symbolic expressions for the goals. A methodology and a tool called *Conjver* to address this question while avoiding prohibitive BDD growth is discussed in Kaivola (2005).

Reachable-State Invariants

The example circuits so far have not required any initialization. They can be brought up in an arbitrary state and still work correctly. For verification based on symbolic simulation, this is the optimal scenario, matching the maximally unrestricted start

state of the simulation. Reset-free circuits are, of course, an anomaly, and most circuits require initialization to bring them to an internally consistent state to guarantee correct behavior. In traditional model checking, the initialization and reachable-state analysis automatically guarantees that only circuit states satisfying such basic consistency are considered, although there is a computational cost for this analysis. In verification by symbolic simulation, on the other hand, accounting for such internal circuit consistency requires human effort. It is done through the manual formulation of consistency invariants and the explicit addition of them to the set of triggers in a simulation.

Consider now the ALU circuit in Fig. 13 with the low power mode enabled, i.e., the parameter *PWRHI* defined as 0. In this circuit, the triggers of the adder correctness property *alu8okB* in Fig. 15 no longer guarantee correct 8-bit adder behavior in a simulation started from an arbitrary state without reset. If the circuit is powered up in a state where *mvldE* is high and no reset happens, *mvldE* continues to stay high until the first MUL operation occurs, because the clock to the *mul* subunit does not toggle. During this time, *mvldE* corrupts every ADD result at the result write-back mux. This does not happen in initialized traces, since reset clears *mvldE*, and it is only going to be set again when there is a valid MUL operation finishing its pipeline, and reset afterwards. Expressed as an internal consistency invariant, the property *mvldEok* in Fig. 20 is missing, even though it is universally valid in every initialized trace of the circuit.

The property *alu8okB* can be augmented by adding an instance of the invariant *mvldEok* to the triggers as in Fig. 21. By Boolean simplification, for example, by parametric substitutions, the triggers together then imply that *mvldE* is 0 in cycle 4, which allows the addition result to flow through the result mux exactly as in Fig. 18.

Just considering the simulation, there is no difference between the new trigger and the previous ones. They are all assumptions restricting the scope of the claim under verification. However, methodologically, the different triggers have distinct roles:

```
property mvldEok;
  @(posedge clk)
    ( mvldE = ( $past(vldA,4) && ( $past(opA,4) == MUL ) ) )
endproperty
assert property(mvldEok);
```

Fig. 20 ALU internal invariant

```
property alu8okC;
  @(posedge clk)
    ( $past(vldA,3) &&
      ( $past(opA,3)==ADD8 ) &&
      ~( $past(vldA,4) && ($past(opA,4)==MUL) ) &&
      ( mvldE = ( $past(vldA,4) && ($past(opA,4)==MUL) ) )
    ) |->
    ( wbW[7:0] == $past(datainB[1][7:0],2) + $past(datainB[2][7:0],2) )
endproperty
```

Fig. 21 Adder correctness property with internal invariant instantiation

– The first and second triggers are inherent to the property being verified: *if ADD8 operation is executed, then result is correct for ADD8*.
– The third trigger is an instance of an external assumption, reflecting an expectation that MUL and ADD8 operations should not execute in patterns that would lead to pipeline hazards.
– The last trigger is an instance of an internal invariant that is expected to be universally true in every initialized trace of the circuit.

An implicit expectation is that the latter two triggers are universally true in the normal operating circumstances of the circuit whenever the basic triggers are true and, therefore, do not really constitute restrictions to the scope of the verification.

This classification of simulation triggers to basic triggers, external assumptions, and internal invariants is ubiquitous in verification based on symbolic simulation, as is the belief that the latter two classes are just auxiliary helpers that do not restrict what is "really" verified. In other words, the verification of the property *alu8okC* of Fig. 21 is expected to imply the original property *alu8ok* of Fig. 14, with only the first two triggers. The validity of this expectation naturally depends on whether the purported invariants are truly invariants. For internal invariants, the best practice is to formally verify them by means of a traditional model checker. The validity of the external assumptions is usually outside the scope of a formal effort and is often done by including the assumptions as simulation checkers in a traditional test environment.

There is a human cost to identify and formulate the internal invariants which over-approximate the reachable-state space of the system to the extent that is necessary to enable the verification of the main validity goals, as well as to determine the exact times at which these invariants should be instantiated as triggers in the symbolic simulation task. There is also a computational cost in the validation of these invariants through external means. The tradeoff of these costs is that they obviate the cost of computing the reachable-state fixed point that traditional model checking would need, enabling the method to handle substantially larger systems. The number and type of internal invariants needed depends on the type of the design and the property to be verified; however, in general, the cost tends to increase with the number of feedback loops present in the circuit. For many straight pipeline designs, most internal invariants needed are akin to the *mvldEok* invariant in Fig. 20 above, relating internal control signals to events at the circuit interface.

Complexity Management

Simulation Complexity

Both symbolic and traditional simulators are often implemented as event-driven simulators, where the computational cost of the simulation is primarily relative to the number of signal value changes and is less sensitive to the number of signals in the circuit. In symbolic simulators, the most important computational

aspect is the construction and management of the symbolic expressions in the simulation. The size of these expressions is almost always the limiting factor in what can be feasibly accomplished. Compared to the computational cost of symbolic expressions, simulation with 0s, 1s, and Xs is practically free. Because of this, it is important to associate symbolic variables with as few signals and times in the stimulus as possible.

For example, in the 8-bit adder circuit trace in Fig. 8, the stimulus associates symbolic variables with the input data signals only in cycle 2, when the operation instance under verification samples them. In all the earlier and later cycles, the data signals have X's. As a result of this, there is only one wave of symbolic data flowing through the circuit in the simulation, one pipestage per cycle.

As a point of comparison, consider the same verification task in a traditional reachable-state or bounded model checker setting. In both cases, the checker would start from the set of initial states, in this case the set of all states, and begin constructing the cross product of the circuit's state space with an automaton encoding the property. This automaton would keep track of three ADD8 operations, one starting in the current cycle, one that started a cycle ago, and one that started two cycles ago. At each state of the cross product, the checker would also keep track of the data values for the three stages of the pipeline.

Contrast this with what happens in symbolic simulation. In cycle 2, symbolic values are only associated with the input data vectors; in cycle 3, only with the internal sum calculation; and in cycle 4, only with the result signals. In each cycle, the data values in the pipestages that do not matter for the one ADD8 instance under focus are abstracted away by the undefined value X. The facts that only one ADD8 instance needs to be tracked and that in each cycle symbolic data is only associated with the pipestage that is relevant for that instance give symbolic simulation a significant performance edge over traditional model checking. Notice also that the abstraction of internal values by Xs happens automatically in the simulation based on the decision to assign symbolic data values to the input signals just for one cycle.

Since the size of the symbolic expressions in the simulation is the primary capacity barrier, it is important to use symbols in the stimulus sparingly and to use the triggers to the simulation to narrow the path that needs to be computed. It is also very important to avoid the computation of symbolic values unnecessarily, in contexts where they do not contribute meaningfully to the verification goal. In a forward simulator, this is not trivial. When simulating a certain cycle, it is not known yet which signal values in that cycle will matter to the verification goal in a later cycle. We look next at techniques for analyzing the progress of symbolic computations and for ways to guide the simulator to avoid or reduce certain symbolic computations.

One particularly straightforward technique for reducing the set of signals for which simulation needs to compute values is the cone of influence (COI) reduction. The validity of the verification goal can only depend on the transitive fan-in of signals referenced in the goal, and therefore, there is no need to simulate any signals outside of this set. All symbolic simulators the authors are aware of apply this technique automatically.

Complexity Analysis

The limits of computational capacity are the limits between what can and what cannot be verified in practice. When attempting to resolve a computational capacity challenge, the most crucial difference between symbolic simulation and traditional model checking is that in symbolic simulation, a computational capacity problem is virtually always extremely concrete. It manifests itself as a symbolic expression that is too large, an expression that is associated with a particular node and time in the simulation. This concreteness allows a human user to analyze, understand, and resolve the problem with a degree of precision that is simply not available in any other verification method that the authors are aware of. This amenability to precise performance analysis is one of the key differentiators enabling the success of symbolic simulation as a verification method.

Performance analysis is further assisted by the fact that the verification focuses on exactly one instance of the property under verification in a symbolic trace. Not only does a performance problem actualize in some specific signal at a specific time, but the human verifier can also understand the role of that signal and time relative to the property and the pipeline. Returning to the ADD8 example, consider the ADD8 operation in the ALU circuit in Figs. 11 and 13, which includes the *mul* unit. If the user attempts to simulate using the triggers in Fig. 21, the simulation may not be completed successfully because the symbolic expressions generated during its process grow too large. Assuming that the simulator has an ability to fail gracefully, the user could then locate sample signals and times at which the large symbolic expressions occur and find out that they are in the *mul* sub-circuit. Based on the user's knowledge about the circuit and the property, it could then be concluded that this computation is simply unnecessary for the verification goal. The techniques to guide the simulator to avoid such computations are discussed below.

The symbolic expressions flowing in the simulation are based on the symbolic variables associated with specific signals and times in the stimulus. These variable dependencies also help the user to understand what the circuit is attempting to compute when there is a problem in the computation. For example, while for ADD8 the symbolic expressions related to the bit-vector addition are small enough that practically any symbolic representation is able to handle them, for ADD16, this is no longer the case, and a good variable ordering is a must for a succinct BDD representation. If this fact had not been discovered yet, and an ADD16 simulation was attempted, prohibitively large expressions would occur in the adder datapath. This scenario is different from the expression grown in the MUL signals above, as now the problem signals are in the datapath for the operation under verification, and they need to be computed to carry out the verification. Looking closer at the expressions, the user would find out that they depend on symbolic variables associated with the input data and that the expression size grows rapidly when going from lower to higher bits in the datapath. This would lead the user to look at the operation the data path is intended to compute, addition, and draw the lesson from general knowledge about BDD behavior that the input data bits should be interleaved in the variable ordering.

The expression size and variable dependency analysis is not only a feature of BDD-based simulation. The authors have found these techniques extremely useful also when the simulation is done with AIGs and the verification goal checking is done using SAT. In practice, there appears a strong correlation between the SAT solver performance and the size of the simulation expressions used as its inputs. This allows the user to identify signals with large expressions, trace the variable dependencies, and determine to what extent the expressions are expected to contribute to the property under verification.

Weakening

The techniques for guiding the simulator to discard or not compute values for certain signals and times during the simulation are collectively called *weakening*. There are three main types of weakening:

1. Universal weakening
2. Cycle-specific weakening
3. Dynamic weakening

All these share the basic idea that at certain points in the simulation, the simulator should replace a value it would otherwise compute with the undefined value X. Since the value X abstracts any value, weakening is safe in the sense that any property that is true over the weakened trace would also be true of the trace that would be computed without the weakening. The inverse is not true: The failure of a property over a weakened trace may be caused by the weakening itself and does not necessarily mean that the property is not valid.

In universal weakening, the user instructs the simulator to discard the values for a certain set of signals across all times in the simulation and replace them with Xs. It is equivalent to the concepts of "free" or "stop-at" present in many model checkers. Effectively the fan-in logic for a weakened signal is discarded, and it becomes a new, unrestricted input to the system. The behaviors of the weakened system naturally over-approximate the original system. For example, in the ADD8 example in the ALU circuit of Fig. 13, with the *mul* subunit, the input data signals to the *mul* subunit could be universally weakened to prevent expression growth there.

Cycle-specific weakening is a more fine-grained version of universal weakening. It allows the user to discard the values for a given set of signals, but instead of all times only for a certain cycle or cycles, like a cycle-specific stop-at directive. This technique is unique to symbolic simulation. The fact that it is even meaningful to talk about signals at specific times in the verification task is directly related to the fact that symbolic simulation focuses on just one fixed instance of the verification goal.

For example, consider again the ADD8 operation in the ALU circuit in Figs. 11 and 13, and assume that the circuit is augmented with a bypass loop back from the

result to the inputs. When simulating the result write-back cycle, the addition result on its way to the circuit output also loops back to the unit inputs through the bypass, causing possible symbolic expression growth both in ADD and MUL. The issue with the MUL can be resolved as above, by universally weakening its data inputs. However, ADD inputs cannot be universally weakened, as the data for the ADD8 operation flows through these signals two cycles earlier. What can be done instead is to weaken these signals at all other cycles than the one needed by the wave of data for the ADD8 operation.

Cycle-specific weakening is an extremely versatile technique that allows the users to apply their intuition about the usage of signals at times relative to the progress of the operation under verification in the reduction of the simulation cost. A common usage model is to weaken a signal at a time and simultaneously associate a fresh symbolic variable with it in the stimulus. Various algorithmic techniques to automatically compute cycle-specific weakening sets based on the verification goals, the circuit, and user intuition about the design intent are also frequently applied.

The third technique, dynamic weakening, is based on looking at expressions as they are computed in the simulation instead of the criteria fixed beforehand. In its most basic form, the user can instruct the simulator to discard any symbolic value and replace it with an X if the expression size for the value exceeds a user-given threshold. In more advanced forms, the user can apply more fine-grained thresholds that may depend on the simulation cycle, the signal name or hierarchy, or the presence or absence of certain symbolic variables in the expression. Dynamic weakening is a very robust technique that in many instances allows the user to quickly resolve complexity issues caused by the computation of unnecessary expressions in the simulation, without the need of more detailed analysis. Dynamic weakening works especially well when there is a reasonable upper bound on the sizes of the expressions needed for the verification goal. For example, in ADD8 verification in the ALU circuit, all the issues with expression growth in the *mul* subunit or through the bypass could be resolved by setting a simple dynamic weakening limit just above the relatively small expression size needed by the adder.

The concreteness of symbolic simulation gives the verification engineer fine-grained visibility into the computations on the level of individual signals, enabling precise analysis and mitigation of computational complexity bottlenecks through weakening. However, the determination of which signals at which simulation times are really needed for a specific verification goal is often a time-consuming task. In many cases this task can be automated by the technique of timed causal fanin analysis (Kaivola and Bar Kama 2022). This method is based on the use of information from a preliminary, more approximate, symbolic simulation run done with a low dynamic weakening threshold, to compute a tight fanin cone of interest for the verification goals, weakening all other circuit logic in the main symbolic simulation run. The method combines the principles of cone-of-influence (COI) reduction and constant-based model reduction on a cycle-by-cycle basis.

Verification Flow

Formal verification based on symbolic simulation is at heart an iterative human-guided flow that starts from the verification goal and minimal stimulus and incrementally augments the stimulus and verification triggers until the goal is verified or a meaningful failure scenario is constructed. Conceptually, the verifier incrementally builds an abstraction of the circuit's behavior using undefined *X* values and symbolic variables. In practice, the verifier repeatedly runs the simulator, observes an undefined *X* value on some signal of interest, analyzes the path that leads to this *X* in the simulation, and strengthens the stimulus or the triggers to resolve the undefined value. Once undefined values are no longer present in the verification goals, the verifier further analyzes the symbolic expressions characterizing any failure conditions to determine whether they might be resolved by further strengthening of the triggers or present a genuine failure of the verification goals.

The debug process is very concrete, based on a strong operational intuition about the flow of values in the simulation. Expression dependencies, i.e., the sets of symbolic variable names present in expressions in the simulation, make this flow highly visible. During the verification process, the verifier inevitably gains a detailed understanding of the flow of computation of the circuit, as this is required to decide which of the many possible ways to resolve an undefined value would be most productive. A guiding principle in this effort is to use symbolic variables in the stimulus as sparingly as possible and to minimize the circuit scope that contributes to the verification goals.

For example, returning to the ALU circuit in Fig. 13 of section "Simulation Scope Control", a first attempt to verify the adder correctness property *alu8ok* in Fig. 14 would lead to an undefined *X* value on the result bus *wbW*. Out of the four inputs to the bus, two have *X*s, the control signal *mvldE* and the internal data bus *prdE*. User insight would be needed to realize that the control signal *mvldE* plays a more important role and that the likely design intent is that *mvldE* is 0 when the result from the adder is routed to the output bus, leading to the addition of an extra trigger as in the property of Fig. 15.

To ease the verification process, the verification often starts with an intentionally overconstrained setup, usually by including an initial reset in the stimulus, turning off any clock-gating logic, or by disabling some functionality of the circuit. In the *alu8ok* example in Fig. 14, the verification would typically start with a stimulus that asserts the reset initially, asserts the valid signal *vldA* just once for the operation of interest, and deasserts it in all other cycles, as in Fig. 22. This simplification allows the verifier to focus first on just the datapath of the ADD8 operation, without possible interferences from other simultaneously executing operations. The goal at this intermediate stage of the work is not to provide complete coverage but to expose the most essential parts of the circuit for the verification goal and to help familiarize the verifier with the circuit. The look-and-feel of the work at this stage is somewhat similar to bounded model checking (BMC) for a fixed bound in the

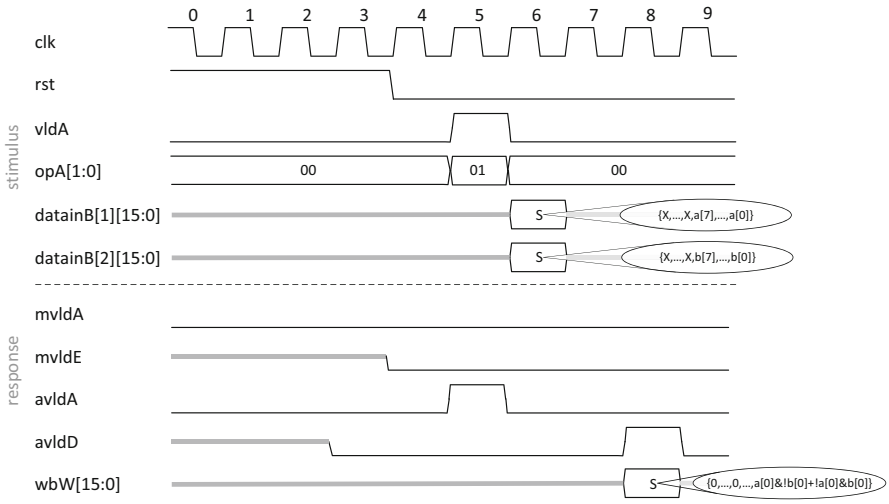


Fig. 22 ALU waveform with initial reset

more conventional formal tools. However, there are essential differences. First, in symbolic simulation, the variable names occurring in expressions give direct insight into the flow of values in the simulation. Second, environment assumptions are not automatically included in symbolic simulation and must be explicitly added to the triggers of the simulation by the user to take effect. Environment assumptions are not automatically instantiated in symbolic simulation, since they can be a major source of computational complexity, alongside circuit scope, and user control is essential in controlling and containing this complexity.

Once stimulus for a successful simulation in an overconstrained environment has been constructed, the overconstraints are gradually removed and replaced by reachable-state restrictions reflecting the expected real operating environment of the design. For example, in verification of the *alu8ok* property in Fig. 14, the user might first remove the restriction that *vldA* is asserted just once, allowing other possibly interfering operations to take place. If only the original two triggers of *alu8ok* are present, in this first step, the user would observe an undefined *X* value in the result write-back bus, trace it through the result write-back mux to a control signal coming from the multiplier, realize that a scheduling constraint between the MUL's and ADD's is needed for proper behavior, and either instantiate a property already present in the model in the trigger or formulate the needed scheduling constraint and then instantiate it, as in property *alu8okB* in Fig. 15. In the second step, the user would remove the initial reset and enable the low power mode, leading to a fully general stimulus. In this stage, the user would notice the need for an internal invariant between the *mvldE* signal and the interface control signals as in Fig. 20 and add the invariant to the trigger, as in property *alu8okC* of Fig. 21.

The simulation X values are a powerful means for detecting design bugs caused by unintentional interferences. In the example system, the X values could be root-caused to missing external or internal invariants. Equally well, the debug process could lead to an observation that the result of an operation can be corrupted by another simultaneously executing operation, a design bug. The strengthening of the stimulus and the trigger primarily takes place through identification of restrictions that are strong enough to narrow the set of behaviors to exclude the problematic propagation of values while being weak enough that the verifier still expects them to hold in all properly initialized states. Human insight is needed in the loop to identify and articulate the reachable-state invariants. The verification done by the simulation is, of course, only as strong as these restrictions, and therefore, it is important that they are well validated, for example, by including the restrictions as run-time checkers in the dynamic simulation environment for the design or by verifying the internal invariants by other formal means.

Arithmetic Circuits

Direct Verification

Most published expositions of symbolic simulation usage in arithmetic verification focus on the hard cases, complex operations that require sophisticated techniques and human ingenuity. Yet, the overwhelming majority of arithmetic operations commonly implemented in hardware can be fully verified with direct symbolic simulation with little user guidance or expertise needed: bit-vector manipulations, shifts, rotates, selections, integer comparisons, integer addition and subtraction, and so on.

The reasons for this are twofold. First, as discussed above, symbolic simulation enables the verifier to efficiently separate the datapath relevant to a single instance of a single operation from the datapaths of all the other operations implemented by the same circuit and all the other instances of operations. This is a significant advantage. Even near-trivial datapaths, such as bitwise-OR, which on their own pose no problem to any serious verification engine, become a challenge when the simple datapath is interwoven with others and a tool is left on its own devices in figuring out that the logic on the simple path is the only thing that really matters for the verification goal.

Second, for large classes of arithmetic operations, there are efficient, concise BDD-based symbolic representations for the values computed in their datapaths. The key factor enabling this is the existence of good variable orderings. For each operation, there tends to be a variable ordering that in practice works for any design implementing the operation, largely irrespective of the implementation choices in the datapath.

To generate a good variable ordering for an operation, the human verifier needs to be able to analyze and anticipate what a BDD representation for the operation

would look like and how a particular variable ordering would affect it. In practice, most BDDs tend to be too large to be intuitively grasped by just visualizing them as graphs. Instead, the authors have found it a useful conceptual tool to consider the BDD as a machine that reads an assignment of values to the symbolic variables in a linear sequence according to the variable ordering and produces the result of the function for that assignment. After having read the assignments to the first n symbolic variables, the machine must retain enough information about the values already read to properly compute the function for all possible assignments to the symbolic variables not yet read. The less information that needs to be retained about the values already read, the more concisely the machine can be represented as a BDD. Although this conceptual analysis is imprecise, it often allows us to estimate the magnitudes of different BDD presentations effectively.

For example, consider the equality comparison $a = b$ between two symbolic vectors $a[n:0]$ and $b[n:0]$. Assume first that the variable ordering is $a[n], \ldots, a[0], b[n], \ldots, b[0]$, placing all variables in vector a above vector b. Consider the state of the "BDD machine" computing $a = b$ after reading the assignment for all the bits $a[n:0]$ but none of the bits $b[n:0]$ yet. At this point, the machine must remember the value for each $a[i]$ in order to compare it to $b[i]$, yet to be read, i.e., it must have at least 2^n different states, implying that the size of the BDD representation for $a = b$ is exponential in n.

Assume then that the variable ordering is interleaved as in $a[n], b[n], \ldots, a[0], b[0]$. Consider the 'BDD machine' computing $a = b$, and the state of the machine after reading the assignment for variables $a[n], b[n], \ldots, a[i], b[i]$ for any i. At this point, the machine only needs to be distinguish two scenarios: either $a[n:i] = b[n:i]$, i.e., a and b agree for all bits above i, or they don't. In the first case, it is possible that $a = b$ and the remaining bits need to be read to see if this is actually true, and in the second case, it is already known that $a = b$ is not true. As for any i, only two states need to be distinguished, the BDD representation of $a = b$ is linear in the number of bits n.

For large classes of operations, good variable orderings are already known from previous experience. For example, for most bit-vector operations, comparisons and addition and subtraction type operations, a good ordering interleaves the symbolic data bits from the different inputs, starting from the most significant bit and ending with the lowest. For operations that use one input as a pointer to manipulate another input, such as logical and arithmetic shifts, rotates, and selects, a good ordering places the pointer input symbolic bits above the bits for the input that is manipulated.

The BDD complexity of an operation and the effects of different variable orderings are best analyzed using the abstract specification of the operation, not an implementation. This allows the user more freedom to experiment, compute the results step by step, and pinpoint the specific stage where adverse expression growth may happen. The authors' empirical observation has been that a variable ordering that is good for an abstract specification is almost universally also good for any of its implementations.

Floating-Point Operations

Let us briefly recall some basics of binary floating-point numbers, a binary representation for a subset of real numbers. A floating-point number can be viewed a triple (s, e, m), where the *sign* s is a single bit, the *exponent* e is an integer, and the *mantissa* m is a fixed-point number. The real number encoded by the triple is $(-1)^s * 2^{e-ebias} * m$, where $ebias$ is some fixed value called the *exponent bias*. In a hardware implementation, both the exponent and the mantissa are represented by bit vectors of some fixed sizes, with the binary point in the mantissa in some fixed position.

The IEEE standard on floating-point numbers (IEEE standard for binary floating-point arithmetic 1985) defines several different representations for them, differing on details, but all adhere to the general idea described above. The standard also defines special encodings for zeros, infinities, and various other exceptional values. We call a floating-point number *normal* if it is not one of these special cases and if the mantissa bit to the left of the binary point is 1, i.e., if m encodes a value that is at least 1 but less than 2. If a floating-point number is not normal, the process of transforming it into an equivalent normal form is called *normalization*.

Since only a small subset of the reals are representable as floating-point numbers, not all results of arithmetic operations on floating-point numbers can be expressed precisely as floating-point numbers themselves. Therefore, the IEEE standard defines the concept of *rounding*, determining which sufficiently close representable number should be used, if the accurate result is not representable.

The abstract specifications of virtually all floating-point operations have the following pattern:

- Compute precise, unrounded result on the basis of the inputs. In theory this would be the infinitely precise result; However, in practice, a sufficiently precise approximation is enough.
- Normalize the unrounded result.
- Round the normalized result.

In a hardware implementation, the stages may not have clear boundaries. In the authors' experience, for most operations, the exponent datapaths computing the unrounded result tend to have only calculations that have manageable BDD behavior such as additions, subtractions, shifts, and so on. Where symbolic complexity problems exist, they are almost always in the mantissa datapath. The normalization stage can be expensive to compute symbolically, and where possible, it is helpful to limit the width of normalization shifts by context-specific information, for example, by knowledge of any bounds that the unrounded mantissa is known to obey. The final stage, rounding, tends to be easy to compute symbolically, although rounding specifications can be tricky to write.

Most unary floating-point operations can be directly verified with symbolic simulation, including, for example, normalization and denormalization, rounding,

reciprocals, conversions between integer and floating-point representations, and so on. Many binary operations are amenable to direct verification as well. For advanced examples of direct verification of operations such as reciprocals and exponentiation, see Bingham and Leslie-Hurd (2014).

Floating-Point Addition

The essence of floating-point addition can be expressed by the pseudo-code in Fig. 23. The mantissas are aligned according to the exponent difference and added or subtracted as bit-vectors depending on the signs. This unrounded result is then normalized and rounded. As a special case, if one input is so much larger than the other that the mantissas would not overlap after the alignment, mantissa addition is not needed at all, as the result is always going to be the larger input, or close to it, depending on the rounding mode.

The basic problem in the symbolic simulation of a floating-point adder is the exponent difference-based variable-length shift that takes place before the addition of the mantissas. To contain BDD growth in bit-vector addition, the variables for the addends should be aligned according to their bit position in the addition. However, with a varying shift, the position to which an input bit ends up in the addition is variable, and the variable ordering cannot be optimized. For any fixed alignment, there would be no problem: for example, if the exponents of both inputs are equal, the symbolic variables for the two mantissas could just be interleaved for a good ordering.

```
// inputs: two normal floating-point sign-exponent-mantissa triples (N_s, N_e, N) and (R_s, R_e, R)
if (N_e ≥ R_e)
        ediff = N_e − R_e;
        if ediff is large
                return ( normalize-and-round (N_s, N_e, N ± ε) )
        elsif (N_s = R_s);   // true addition, signs agree
                S = (N ≪ ediff) + R;   // align mantissas and add
                return ( normalize-and-round (N_s, R_e, S) )
        else   // true subtraction, signs opposite
                S = (N ≪ ediff) − R;   // align mantissas and subtract
                if S ≥ 0   // did the sign flip?
                        return ( normalize-and-round (N_s, R_e, S) )
                else
                        return ( normalize-and-round (¬N_s, R_e, −S) )
                endif
        endif
else . . . symmetric with N and R exchanged . . .
endif
```

Fig. 23 Floating-point addition

The solution to this problem pulls together three distinct ingredients. First, since mantissa addition does not need to be done at all when the exponent differences are either too large or too small, mantissa addition matters only for a small finite set of distinct exponent differences. This set can be exhaustively iterated over with a case split, and each exponent difference can be considered separately. Second, for each such fixed exponent difference, there is a good variable ordering, aligning the symbolic variables for the input mantissas according to their position in the addition after the mantissa shift based on the exponent difference. Third, the technique of parametric substitutions makes it possible to factor in the condition fixing the exponent difference and carry out the simulation exactly under the scenario with the given exponent difference, fixing the mantissa alignment shift both in the implementation and the specification. Note, in particular, that in the case split, the input exponents are not fixed themselves, just the difference between them is fixed.

As a further nuance, the true subtraction case, where the two inputs have opposing signs, may lead to a wide normalization shift prior to rounding in cases where the exponents are equal or almost equal, and the mantissas cancel each other out almost completely in the subtraction. As the width of the shift depends on the mantissa values, it is symbolic and not fixed, which may cause symbolic expression growth in the normalization and rounding stages. For certain designs, a second-level case split iterating over the possible widths of the normalization shift and considering each shift width separately may be needed to contain BDD growth. Again, the technique of parametric substitutions is essential in enabling us to carry out the simulation exactly under the scenario with the fixed normalization shift.

For a more detailed floating-point addition verification case study, see Aagaard et al. (1999a).

Integer Multiplication

Most integer multiplication circuits are not directly amenable to verification by symbolic simulation using BDDs, as it is well known that any BDD representation for the middle bits of multiplication is of exponential size in the number of input bits, irrespective of the variable ordering (Bryant 1986). As an exception, 8-by-8 bit multiplication can be easily handled by direct symbolic simulation and 16-by-16 bit almost always with some care and a small case split fixing some input bits.

To work around the symbolic expression size barrier that prevents direct verification, multiplier verification is typically approached through a two-stage decomposition that in the first stage verifies that a set of partial products are properly derived from the inputs and in the second stage that the output of the circuit is computed as a sum of the partial products. In the context of the simple MUL circuit of Fig. 12, if we write $d1$ and $d2$ for the input values of the circuit, $pp[i]$ for partial products and wb for the output, the essence of the verification goals for the two stages can be expressed by the following equations, also expressed as SVA properties in Fig. 24:

```
property mulppok[i];
@(posedge clk)
( $past(vldA,4) && ( $past(opA,4)==MUL ) && ...
) |->
( $past(ppC[i],2) == ( $past(datainB[1],3) << i ) * ( $past(datainB[2][i],3) ) )
endproperty

property mulwbok;
@(posedge clk)
( $past(vldA,4) && ( $past(opA,4)==MUL ) && ...
) |->
( wbW == $past(ppC[15],2) + ... + $past(ppC[0],2) )
endproperty
```

Fig. 24 Decomposed multiplication correctness properties

$$\begin{aligned} pp[i] &= d1 * (d2[i] \ll i) \text{ for all } i \\ wb &= pp[n] + \ldots + pp[0] \end{aligned}$$

In the first stage of the verification, symbolic variables are associated with the input signals in the stimulus, and the symbolic expressions for the partial products are computed through simulation. All operations performed on the way from inputs to partial products are simple and have concise BDD representations.

In the second stage, the partial product signals are weakened, and another set of symbolic variables are associated with them in the stimulus. Due to the weakening, the relation between partial products and the inputs is lost, and the partial products effectively become free inputs in the simulation. The main operations computed between partial products and the result are all additions. A variable ordering interleaving the variables associated with the partial product signals according to their alignment in the addition results in concise BDDs in most cases, allowing the relation between partial products and the output to be verified by direct simulation.

Real hardware multipliers are more complex than the simple example here. Typically, they use Booth or other encodings to reduce the number of partial products and have optimized adder trees (Booth 1951). However, the verification principles are the same: the identification of partial products in the design, and the verification of the relations between the inputs and the partial products, and the partial products and the result separately. The mathematical "ideal" partial product that the decomposed reference model refers to does not necessarily exist in a single signal in the design. Instead, it is a verifier's abstraction of information extracted from the simulation. For example, with Booth encodings, a partial product is often coded as a ones' complement vector plus a negate bit, and these two added together form the mathematical notion of a partial product.

As in the second stage of the verification, the partial product signals are weakened, and the relation between them and the inputs is lost; the verification is more general than strictly needed, covering all partial product signal bit combinations and not only those that are actually realizable in the design. This may lead to spurious verification failures if the logic downstream from the partial products relies on some

restrictions on the partial product signals. To enable verification, such restrictions need to be recaptured with explicit side conditions, which are verified in the first stage and then used as extra assumptions in the second stage. For example, the lowest and highest Booth encoding for an input often do not allow the full range of values, restricting the possible range of the corresponding partial products, and this may need to be captured by a side condition.

For a full-fledged integer multiplier verification example, especially in-depth discussion about theorem proving to relate the decomposed reference model to a simple high-level input-output reference model, see O'Leary et al. (2013). For earlier related work, see Kaivola and Narasimhan (2001).

Floating-Point Multiplication and Fused Multiply-Add

At an abstract level, floating-point multiplication can be computed as in the pseudo-code in Fig. 25. As with most other floating-point operations, most of the verification complexity is in the mantissa datapath for the unrounded result. This mantissa datapath is essentially an integer multiplication datapath, with the result forwarded to further processing for normalization and rounding. The normalization stage does not have significant symbolic complexity as only two normalization cases need to be handled: if the inputs are normal, their mantissas are at least 1.0 and below 2.0, which means that the product mantissa is at least 1.0 and less than 4.0, implying that the product mantissa is automatically either normal or just above normal. The rounding stage does not have significant BDD complexity.

Since the mantissa datapath essentially performs integer multiplication, the decomposition strategy of section "Integer Multiplication" can be applied. Partial products are identified in the design, the relation between inputs and partial products is verified first, and then the relation between the sum of partial products and the output. The latter relation, from partial products to the output, also contains exponent and sign calculation from the primary inputs, as well as a reference model implementation of normalization and rounding. Some complex floating-point multiplier datapaths may need an additional decomposition stage after the partial products but before the output, containing the same value as the sum of

// inputs: two normal floating-point sign-exponent-mantissa triples (N_s, N_e, N) and (R_s, R_e, R)
// product sign
$P_s = N_s$ **xor** R_s;
// product exponent
$P_e = N_e + R_e - ebias$;
// product mantissa (with twice as many fraction bits as the inputs)
$P = N * R$;
return (normalize-and-round(P_s, P_e, P));

Fig. 25 Floating-point multiplication

partial products in a more concise set of signals. For a more detailed floating-point multiplier verification example, see Kaivola and Narasimhan (2001).

Fused multiply-addition (FMA) is an operation that performs a multiplication of two floating-point inputs and the addition of a third input, conceptually infinitely precisely, and then normalizes and rounds the result of the addition, all in one operation. FMA differs from a multiplication and an addition done in sequence in that there is no rounding and subsequent loss of precision between the multiplication and the addition. Many contemporary arithmetic hardware designs have a fused multiply-adder as their basic building block.

The verification of a fused multiply-adder can be done with a combination of the techniques for multipliers and adders. For multiplication, partial products are identified in the design. Another decomposition point after partial products but before the addition of the third input may also be needed. The relations between the inputs and the partial products and the optional second decomposition point are verified as for a multiply operation. Then, from this point on, the output is verified as the addition of the product, or sum of partial products, and the third input, using the same exponent difference-based case split as for floating-point addition. The symbolic complexity of the normalization stage in a fused multiply-adder is higher than in a plain multiplier, as an FMA does not guarantee the same kinds of bounds on the unrounded result that lead to only a few possible normalization scenarios for the plain multiplication. Handling of denormal input or output values also increases complexity. Both aspects may necessitate further case splitting to contain the symbolic complexity of the individual simulations. For more detailed fused multiply-adder verification examples, see Slobodova and Nagalla (2004), Slobodova (2006), and KiranKumar et al. (2012). For a related approach with very similar case splitting considerations, see Jacobi et al (2005).

Floating-Point Division and Square Root

While the authors are not aware of any precise theoretical bounds for binary expression sizes for division along the lines of those for multiplication, experience has shown that expression growth for division is at least as bad as for multiplication, and direct verification by symbolic simulation is feasible for only relatively small operand sizes.

Division differs from all the operations above in two important ways. First, useful abstract reference models are by necessity *relational* instead of *functional*, describing a variety of correct implementations instead of a single one. Second, both the reference model algorithms and implementations are *iterative*, containing a loop computing increasingly precise approximations of the mathematically precise result, which in itself may not even be accurately representable in a finite form. An iterative implementation may either consist of a loop unrolling or the same hardware being used repeatedly over multiple cycles.

Consider the simple Sweeney-Robertson-Tocher (SRT) type iterative division algorithm in Fig. 26, using the redundant quotient digit set $\{-1, 0, 1\}$. It takes two

// inputs: two normal floating-point sign-exponent-mantissa triples (N_s, N_e, N) and (D_s, D_e, D)
// initial quotient and remainder
$Q[0] = 1;\ \ R[0] = N - D;$
// compute the first *imax* quotient fraction bits
for $i = 1$ upto *imax*
    // determine quotient digit q_i
    select $q_i \in \{-1, 0, 1\}$
    // update quotient and remainder
    $Q[i] = Q[i-1] + 2^{-i} * q_i;\ \ R[i] = 2 * (R[i-1] - q_i * D);$
// correct Q if final remainder is negative
$Q = (R[imax] < 0)\ ?\ Q[imax] - 2^{-imax}\ :\ Q[imax];$
// sign and exponent
$Q_s = N_s$ **xor** $D_s;\ \ Q_e = N_e - D_e + ebias;$
return (normalize-and-round(Q_s, Q_e, Q));

Fig. 26 Iterative floating-point division

normal floating-point numbers as inputs and produces the rounded quotient of the first input divided by the second. In the pseudo-code the floating-point exponents N_e and D_e are viewed as integers, and the mantissas N and D as fractions. The number of iterations $imax$ depends on the required precision of the result. Conceptually, enough quotient bits need to be computed to guarantee that the rest of the quotient does not matter for the result after rounding. In practice, this is the target precision mantissa size plus a few bits.

The algorithm does not specify the precise selection of the quotient digits q_i. Any selection will do, as long as the *convergence bound*

$$-D \leq R[i] < D$$

is maintained. The presence of negative quotient digits allows an implementation leeway in over-approximating or under-approximating the true result, as long as the computed quotient approximation does not diverge too far from the true result. In a real hardware implementation, the quotient selection logic is typically the critical path, and the satisfaction of the convergence bound the most intricate correctness requirement. Industrial hardware implementations of division algorithms are often more complex than the simple one above. For example, they may produce a sequence of quotient digits per iteration, use redundant or multiple representations of Q and R, or perform speculative calculations. Nevertheless, the principles are similar, and the verification task is structured the same way.

Divider verification uses an iterative decomposed reference model similar to Fig. 26. In the verification, the sequence of partial quotients and remainders $Q[i]$ and $R[i]$ is mapped to hardware signals and times reflecting the progress of the iterative computation in the circuit. The reference model does not fix the quotient digit q_i. Any legal value q_i is allowed, as long as both $Q[i]$ and $R[i]$ are computed

from $Q[i-1]$ and $R[i-1]$ consistently using the same q_i, and the convergence bound $-D \leq R[i] < D$ is satisfied.

Conformance to the reference model is verified separately for each iteration. For iteration i, the previous quotient and remainder $Q[i-1]$ and $R[i-1]$ are considered free inputs, i.e., weakened and associated with fresh symbolic variables in the stimulus, and the values of the next quotient and remainder $Q[i]$ and $R[i]$ are computed by symbolic simulation. The relations between the previous and next values and the convergence bound, expressed as property formulas, are then verified in the simulation. A downside of the verification of each iteration in isolation is that any implicit restrictions in the previous quotient and remainder representations in the circuit are lost, as they are considered free inputs. When the proper behavior of the circuit logic depends on such restrictions, they need to be explicitly characterized as invariants. The identification and formulation of such side invariants often requires detailed understanding of the implementation.

The symbolic expression complexity in the verification of a single iteration is in most cases easily manageable, as good symbolic variable orderings exist for the addition and subtraction operations in the loop update relation. The hardest aspect is the computation of the next quotient digit q_i and the update of the remainder $R[i]$ as a function of q_i. As q_i itself is a function of the previous remainder $R[i-1]$, the next remainder $R[i]$ depends on $R[i-1]$ in two ways, directly through the update relation and indirectly through q_i. This may set conflicting requirements for symbolic variable positions in the ordering and lead to expression growth. The symbolic expression complexity rises as a function of the width of q_i, i.e., the number of bits computed per iteration. For very high radix dividers, careful techniques may be needed to manage symbolic expression size, for example, case splitting on the next quotient digit q_i value to cut the double dependency of $R[i]$ on $R[i-1]$.

The validation of the reference model, i.e., the question "why do we trust that the reference model computes division" belongs to the domain of theorem proving and is discussed in more detail in Kaivola and Aagaard (2000) and O'Leary et al. (2013). The essence of the argument is that the reference model guarantees the loop invariant

$$N = Q[i] * D + 2^{-i} * R[i]$$

Together with the convergence bound above, this implies that the quotients $Q[i]$ are increasingly accurate approximations of N/D. For a more detailed example of divider verification, see Kaivola and Aagaard (2000).

Floating-point square root can be computed by algorithms very similar to those for division, and the verification approach outlined above extends to square root as well. The update relation is more intricate and any issues in symbolic expression growth tend to be worse for square root than for division. On the other hand, the fact that square root is a unary operation allows direct verification for wider operand sizes.

Industrial Verification

The previous sections enumerated some of the most common types of arithmetic circuits and verification strategies for them. In practice, such datapaths seldom occur in isolation. They are bundled together in an integrated design unit that implements a family of arithmetic operations, often multiplexing different datapaths together to maximize circuit reuse. The verification engineer then typically faces the task of validating all the operations supported by the design. The largest examples of such verification tasks that the authors have been involved in have encompassed the full Execution Cluster EXE in Intel Core™ and Intel Atom® processor designs, as well as the graphics floating-point units in Intel Gen graphics designs.

The management of such large-scale verification tasks poses several methodological challenges. As a verification method, symbolic simulation requires closer user guidance than traditional model checking: decisions on signals and times with which to associate symbolic variables in the stimulus, instantiations of external control assumptions, judicious weakening to contain simulation scope, variable orderings to manage symbolic expression complexity, and so on. In a live development project, this verification collateral needs to be maintained throughout the project timeline and modified to account for design changes. All verification goals also need to be frequently revalidated.

To make large-scale verification tasks possible, the verification environment needs to be highly programmable. In the symbolic simulation verification environment used for most Intel development projects, this is achieved by embedding the core symbolic simulator in a code layer called *relational STE* (*rSTE*) in the context of a full-fledged functional programming language. The common computational complexity reduction techniques discussed above, including weakening, parametric substitution, etc., are made easily accessible to the user through programmable options to the tool. The framework also provides sophisticated debug support, breakpoints, waveform and circuit visualization, etc., to enable the user to quickly focus on usual verification problems. For the verification of the implication between the input and output constraints, the tool uses the methods discussed in Kaivola (2005).

A productive verification environment for large-scale tasks also needs to be able to take advantage of commonality between individual verification goals. This commonality occurs in two dimensions: first, between different operations on the same design, and second, between the same operation on different designs. The first kind is reflected, for example, in the signals and timing of an operation, or control assumption instantiations. The second manifests itself in the overall verification strategies for different types of operations, as outlined in the previous sections.

The existence of reusable tried-and-tested verification strategies or "recipes" for solving particular classes of problems is a key requirement for reliable and timely verification work. Such recipes allow the verification engineers either to identify a computational strategy or to flag a verification task as being of unknown complexity. In practice, the recipes are represented as code artifacts, not just as

abstract guidelines. This code collateral needs to be sufficiently abstract and design-independent that it can be used and maintained across different projects over a long period of time. In the authors' experience, such code collateral often pays its cost back the first time it is reused.

In Intel's verification environment, the reusable collateral for arithmetic verification is captured in a single large software artifact called *Common Verification Environment* (CVE). It is a result of substantial software and proof engineering (Kaivola and Kohatsu 2003) to create a standard, uniform methodology for writing specifications and carrying out verification tasks. The aim of the effort is to support reuse and code maintenance over a continuously changing design and separate common and project-specific parts to allow shared code to be written only once. Programming language paradigms such as algebraic and other structured datatypes are used to enforce abstraction and shared structure. Code collateral and related methodology also covers control invariant, state and bypass verification, and interfaces with mainstream simulation environment for external assumption checking. For a more detailed exposition, see Kaivola et al. (2009).

Related Work

Symbolic simulation as a hardware verification technique has its roots in program verification. The application of program verification technology to hardware designs was first suggested by John Darringer of the IBM Thomas J. Watson Research Center (Darringer 1979). Darringer illustrated the application of the Effigy program verification tool (Darringer and King 1978; King 1979) to verification of gate-level hardware designs and of microcode.

The *practical* application of symbolic simulation to hardware had to wait for an efficient way to represent and manipulate Boolean functions: the binary decision diagram (BDD) of Akers (1978) and most particularly the reduced ordered BDD (ROBDD) of Bryant (1986). The ROBDD made it possible to scale symbolic simulation to realistic transistor-level (MOSSYM (Bryant 1985)) and gate-level circuit designs (Beatty et al. 1990; Bryant 1990).

Symbolic trajectory evaluation (STE) was invented by Carl Seger and Randal E. Bryant as a generalization of symbolic simulation and an alternative to symbolic model checking (Bryant and Seger 1990; Seger and Bryant 1995; Hazelhurst and Seger 1997). STE accepts specifications that are pairs of *trajectory formulas* - describing (conditional) values on signals, conjunction, and "next-time" operator. One formula specifies the stimulus to the circuit, and another specifies the desired response. Symbolic simulation is used as the engine to check the implication. A distinguishing characteristic of STE is its automatic use of abstraction in the form of the undefined value X, which enables very large circuits to be handled. Seger embedded the STE simulator in a special-purpose functional programming language fl, and the resulting *Voss* system (Seger 1993) was used to verify a number of designs including complex arithmetic (Aagaard and Seger 1995).

Carl Seger joined Intel in 1995, bringing the Voss system with him. Voss formed the basis of a major new system called *Forte*. Almost immediately Forte saw application to floating-point arithmetic in the Intel Pentium® Pro processor design project in the aftermath of the Pentium processor FDIV bug of 1994 (Pratt 1995). Most of the fundamental techniques described in the current chapter were developed in this timeframe at Intel's Strategic CAD Labs (SCL): parametric substitutions (Jones 2002; Aagaard et al. 1999a), many case studies and methodology development (Jones et al. 2001; O'Leary et al. 1999), manipulating programs as objects through Lifted FL (Aagaard et al. 1999b) and reasoning about them in the theorem prover ThmTac (Aagaard et al. 1999c). The wave of development as a whole is summarized in the overview paper (Seger et al. 2005).

In the next phase, much of the symbolic simulation-based formal verification work at Intel moved from research groups to dedicated formal verification teams in the product development organizations. This began with the Pentium 4 processor development project in 1999. This work included technical advances in verification of complex circuits such as multipliers (Kaivola and Narasimhan 2001), fused multiply-adders (Slobodova and Nagalla 2004), and iterative algorithms including dividers (Kaivola and Aagaard 2000; Aagaard et al. 2000). Advances were made in verification engineering and the management of large-scale verification tasks in an active project development setting (Kaivola and Kohatsu 2003). The extension of STE to a relational formalism like the one used in this chapter was first outlined in Kaivola (2005), and the overall verification methodology was presented in the capstone paper (Kaivola et al. 2009). Alongside the wide-scale deployment, the verification infrastructure was improved by the adoption of the reFLect functional programming language (Grundy et al. 2006), an evolution of the fl language used by the original Forte. The precise semantics of the relational extension of STE was first formulated in the context of the theorem prover Goaled (O'Leary et al. 2013), tightly connecting symbolic simulation to high-level reasoning.

Outside the area of arithmetic verification, symbolic simulation has been used for verification of embedded memory arrays (Pandey et al. 1996; Krishnamurthy et al. 2000), control logic (Kaivola and Naik 2005; Kaivola 2005), and automatic symbolic indexing abstractions (Adams et al. 2007). Techniques for automatic refinement of the X-abstraction used in STE are discussed in Tzoref and Grumberg (2006) and Roorda and Claessen (2006). More recently, symbolic simulation has been applied to security verification (Bar Kama and Kaivola 2021) and verification of Error Correction Code (ECC) algorithms and implementations (Gupta et al. 2022).

Extensions of symbolic simulation include word-level methods (Chakraborty et al. 2017). Also, a formalism and tool called *generalized STE* (GSTE) (Yang and Seger 2003) extends symbolic simulation with mechanisms to handle feedback loops.

The *Handbook of Model Checking* contains a precise, but very readable, introduction to the theory underlying symbolic trajectory evaluation (Melham 2018). General treatments of symbolic simulation in practice can be found in the monographs by Jones (2002) and Bertacco (2006).

A symbolic simulator has been developed and integrated within the ACL2 theorem prover (Swords 2010, 2017; Swords and Davis 2011; Slobodova et al. 2011). The tool supports bit-level and word-level symbolic simulation of both software and hardware with BDD and AIG representations and implements many of the paradigms discussed in this chapter. It has been extensively used in industrial verification (Goel et al. 2021). The simulator is available in open source as part of ACL2.

The Intel tool Forte described in this chapter is proprietary; however, an evolution of both Voss and Forte called VossII now exists in open source (VossII 2020).

Acknowledgments This chapter summarizes almost three decades of work. At the time of its writing, more than 60 people have directly contributed to Intel's shared arithmetic verification code-base on over 50 design projects. A much higher number have contributed intellectually to the endeavor, both inside Intel and at large. It has been a team effort. We would like to express our sincere thanks to each and every member of the team.

We would also like to thank Intel's design and validation management over the years for trusting and encouraging this work. There are more names than we can mention; however, we would like to express our special thanks to Bob Bentley and Alon Flaisher for their crucial long-term support.

Finally, we would like to thank Jesse Bingham, Levent Erkok, Robert Jones, Joe Leslie-Hurd, Sayak Ray, and Annette Upton for their detailed feedback on the drafts of this text.

References

Aagaard M, Seger C-J (1995) The formal verification of a pipelined double-precision IEEE floating-point multiplier. In: Proceedings of IEEE international conference on computer aided design (ICCAD), pp 7–10

Aagaard MD, Jones RB, Seger C-JH (1999a) Formal verification using parametric representations of Boolean constraints. In: Proceedings of the 36th annual ACM/IEEE design automation conference, pp 402–407

Aagaard MD, Jones RB, Seger C-JH (1999b) Lifted-FL: a pragmatic implementation of combined model checking and theorem proving. In: Theorem proving in higher order logics. Springer, pp 323–340

Aagaard MD, Melham TF, O'Leary JW (1999c) Xs are for trajectory evaluation, Booleans are for theorem proving. In: Pierre L, Kropf T (eds) Correct hardware design and verification methods. Springer, pp 202–218

Aagaard MD, Jones RB, Kaivola R, Kohatsu KR, Seger C-JH (2000) Formal verification of iterative algorithms in microprocessors. In: Proceedings of the 37th annual design automation conference, pp 201–206

Adams S, Bjork M, Melham T, Seger C-J (2007) Automatic abstraction in symbolic trajectory evaluation. In: Formal methods in computer aided design (FMCAD'07), pp 127–135

Akers SB (1978) Binary decision diagrams. IEEE Trans Comput C-27:509–516

Bar Kama N, Kaivola R (2021) Hardware security leak detection by symbolic simulation. In: 2021 formal methods in computer aided design (FMCAD), pp 34–41

Beatty DL, Bryant RE, Seger C-JH (1990) Synchronous circuit verification by symbolic simulation: an illustration. In: Proceedings of the sixth MIT conference on advanced research in VLSI, pp 98–112

Bertacco V (2006) Scalable hardware verification with symbolic simulation. Springer

Bingham JD (2015) Universal Boolean functional vectors. In: Formal methods in computer-aided design (FMCAD), pp 25–32

Bingham J, Leslie-Hurd J (2014) Verifying relative error bounds using symbolic simulation. In: Biere A, Bloem R (eds) Proceedings of the 26th international conference on computer aided verification (CAV 2014). Lecture notes in computer science, vol 8559. Springer, pp 277–292

Bjesse P, Boralv A (2004) DAG-aware circuit compression for formal verification. In: IEEE/ACM international conference on computer aided design, 2004. ICCAD-2004, pp 42–49

Booth AD (1951) A signed binary multiplication technique. Q J Mech Appl Math 4:236–240

Bryant RE (1985) Symbolic verification of MOS circuits. In: 1985 Chapel Hill conference on VLSI, pp 419–438

Bryant RE (1986) Graph-based algorithms for Boolean function manipulation. IEEE Trans Comput C-35:677–691

Bryant RE (1990) Verification of synchronous circuits by symbolic logic simulation. In: Hardware specification, verification and synthesis: mathematical aspects. Lecture notes in computer science, vol 408. Springer, pp 14–24

Bryant RE, Seger C-JH (1990) Formal verification of digital circuits using symbolic ternary system models. In: International conference on computer aided verification, pp 33–43

Chakraborty S, Khasidashvili Z, Seger C-JH, Gajavelly R, Haldankar T, Chhatani D, Mistry R (2017) Symbolic trajectory evaluation for word-level verification: theory and implementation. Formal Methods Syst Des 50(2–3):317–352

Darringer J (1979) The application of program verification techniques to hardware verification. In: 16th design automation conference, pp 375–381

Darringer J, King JC (1978) Applications of symbolic execution to program testing. IEEE Des Test Comput 51–60

Drane T, Kiran Kumar MA (2022) C-to-RTL equivalence checking. In: Chattopadhyay A (ed) Handbook of computer architecture. Springer

Goel S, Slobodova A, Sumners R, Swords S (2021) Balancing automation and control for formal verification of microprocessors. In: Silva A, Leino KRM (eds) Computer aided verification. Springer, pp 26–45

Grundy J, Melham T, O'Leary J (2006) A reflective functional language for hardware design and theorem proving. J Funct Program 16(2):157–196

Gupta A, Kaivola R, Mehta M, Singh V (2022) Error correction code algorithm and implementation verification using symbolic representations. In: Griggio A, Rungta N (eds) Formal methods in computer aided design (FMCAD), pp 151–159

Harrison J (2009) Handbook of practical logic and automated reasoning. Cambridge University Press

Hazelhurst S, Seger C-JH (1997) Symbolic trajectory evaluation. In: Kropf T (ed) Formal hardware verification: methods and systems in comparison. Springer, pp 3–78

IEEE standard for binary floating-point arithmetic (1985) Institute of Electrical and Electronics Engineers. Note: Standard 754–1985

IEEE standard for SystemVerilog–unified hardware design, specification, and verification language (2018) IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012), pp 1–1315

Jacobi C, Weber K, Paruthi V, Baumgartner J (2005) Automatic formal verification of fused-multiply-add FPUs. In: Proceedings of the conference on design, automation and test in Europe – Volume 2, DATE'05. IEEE Computer Society, pp 1298–1303

Jones RB (2002) Symbolic simulation methods for industrial formal verification. Springer

Jones RB, O'Leary JW, Seger C-JH, Aagaard MD, Melham TF (2001) Practical formal verification in microprocessor design. IEEE Des Test Comput 18(4):16–25

Kaivola R (2005) Formal verification of Pentium 4 components with symbolic simulation and inductive invariants. In: Etessami K, Rajamani SK (eds) Computer aided verification. Springer, pp 170–184

Kaivola R, Aagaard M (2000) Divider circuit verification with model checking and theorem proving. In: Theorem proving in higher order logics. TPHOLs 2000. Lecture notes in computer science, vol 1869. Springer, pp 338–355

Kaivola R, Kohatsu KR (2003) Proof engineering in the large: formal verification of Pentium 4 floating-point divider. Int J Softw Tools Technol Transf 4(3):323–334

Kaivola R, Naik A (2005) Formal verification of high-level conformance with symbolic simulation. In: Tenth IEEE international high-level design validation and test workshop, 2005, pp 153–159

Kaivola R, Narasimhan N (2001) Formal verification of the Pentium 4 multiplier. In: Proceedings sixth IEEE international high-level design validation and test workshop, Los Alamitos. IEEE Computer Society, pp 115–120

Kaivola R, Ghughal R, Narasimhan N, Telfer A, Whittemore J, Pandav S, Slobodová A, Taylor C, Frolov V, Reeber E, Naik A (2009) Replacing testing with formal verification in Intel Core i7 processor execution engine validation. In: Bouajjani A, Maler O (eds) Computer aided verification. Springer, pp 414–429

Kaivola R, Bar Kama N (2022) Timed causal fanin analysis for symbolic circuit simulation. In: Griggio A, Rungta N (eds) Formal methods in computer aided design (FMCAD), pp 99–107

King JC (1979) Symbolic execution and program testing. Commun ACM 19(7):385–394

KiranKumar VMA, Gupta A, Ghughal R (2012) Symbolic trajectory evaluation: the primary validation vehicle for next generation Intel processor graphics FPU. In: 2012 formal methods in computer-aided design (FMCAD), pp 149–156

Krishnamurthy N, Martin AK, Abadir MS, Abraham JA (2000) Validating PowerPC microprocessor custom memories. IEEE Des Test Comput 17(4):61–76

Kuehlmann A, Paruthi V, Krohm F, Ganai M (2002) Robust Boolean reasoning for equivalence checking and functional property verification. IEEE Trans Comput-Aided Des Integr Circuits Syst 21(12):1377–1394

Melham T (2018) Symbolic trajectory evaluation. In: Clarke EM, Henzinger TA, Veith H, Bloem R (eds) Handbook of model checking, ch. 25. Springer, pp 831–870

O'Leary J, Zhao X, Gerth RT, Seger C-JH (1999) Formally verifying IEEE compliance of floating-point hardware. Intel Tech J, pp 1–14

O'Leary J, Kaivola R, Melham T (2013) Relational STE and theorem proving for formal verification of industrial circuit designs. In: 2013 formal methods in computer-aided design, pp 97–104

Pandey M, Raimi R, Beatty DL, Bryant RE (1996) Formal verification of PowerPC arrays using symbolic trajectory evaluation. In: DAC'96: proceedings of the 33rd annual design automation conference. Association for Computing Machinery, pp 649–654

Pratt V (1995) Anatomy of the Pentium bug. In: Mosses PD, Nielsen M, Schwartzbach MI (eds) TAPSOFT'95: theory and practice of software development. Springer, pp 97–107

Ray S, Goel S (2022) Theorem proving. In: Chattopadhyay A (ed) Handbook of computer architecture. Springer

Roorda J-W, Claessen K (2006) SAT-based assistance in abstraction refinement for symbolic trajectory evaluation. In: Ball T, Jones RB (eds) Computer aided verification. Springer, pp 175–189

Russinoff DM (1998) A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor. LMS J Comput Math 1:148–200

Russinoff D (2019) Formal verification of floating-point hardware design. Springer

Seger C-JH (1993) Voss – a formal hardware verification system user's guide. https://doi.org/10.5555/901942

Seger C-JH, Bryant RE (1995) Formal verification by symbolic evaluation of partially-ordered trajectories. Formal Methods Syst Des 6:147–190

Seger C-J, Jones R, O'Leary J, Melham T, Aagaard M, Barrett C, Syme D (2005) An industrially effective environment for formal hardware verification. IEEE Trans Comput-Aided Des Integr Circuits Syst 24(9):1381–1405

Seligman E, Schubert T, Kiran Kumar MVA (2015) Formal verification: an essential toolkit for modern VLSI design. Morgan Kaufmann Publishers Inc.

Slobodova A (2006) Challenges for formal verification in industrial setting. In: FMICS'06/PDMC'06: Proceedings of the 11th international workshop, FMICS 2006

and 5th international workshop, PDMC conference on formal methods: applications and technology. Springer, pp 1–22

Slobodova A, Nagalla K (2004) Formal verification of floating point multiply add on Itanium processor. In: Fifth international workshop on designing correct circuits, ETAPS 2004

Slobodova A, Davis J, Swords S, Hunt W (2011) A flexible formal verification environment for industrial scale verification. In: Ninth ACM/IEEE international conference on formal methods and models for codesign (MEMOCODE2011), pp 89–97

Swords SO (2010) A verified framework for symbolic execution in the ACL2 theorem prover. PhD thesis, University of Texas at Austin. http://hdl.handle.net/2152/ETD-UT-2010-12-2210

Swords S (2017) Term-level reasoning in support of bit-blasting. In: Slobodova A, Hunt WA Jr (eds) Proceedings 14th international workshop on the ACL2 theorem prover and its applications, Austin, 22–23 May 2017. Electronic proceedings in theoretical computer science, vol 249. Open Publishing Association, pp 95–111

Swords S, Davis J (2011) Bit-blasting ACL2 theorems. In: Hardin D, Schmaltz J (eds) Proceedings 10th international workshop on the ACL2 theorem prover and its applications, Austin, 3–4 Nov 2011. Electronic proceedings in theoretical computer science, vol 70. Open Publishing Association, pp 84–102

Tzoref R, Grumberg O (2006) Automatic refinement and vacuity detection for symbolic trajectory evaluation. In: Ball T, Jones RB (eds) Computer aided verification. Springer, pp 190–204

Vizel Y, Ivrii A (2022) Bit level model checking algorithms. In: Chattopadhyay A (ed) Handbook of computer architecture. Springer

VossII (2020). https://github.com/TeamVoss/VossII. Accessed: 8 June 2021

Yang J, Seger C-J (2003) Introduction to generalized symbolic trajectory evaluation. IEEE Trans Very Large Scale Integr (VLSI) Syst 11(3):345–353

Microprocessor Assurance and the Role of Theorem Proving

37

Shilpi Goel and Sandip Ray

Contents

Introduction 1322
ACL2 Preliminaries 1324
Logic Basics 1325
Extension Principles 1327
The Theorem Prover 1328
Some Execution Features: Guards, `MBE`, and `Stobjs` 1329
ISA Analysis 1332
ISA Formalization 1333
Mechanical Analysis for ISA 1334
Binary Code Analysis with ISA Models 1335
Some Formalized ISAs 1336
Analysis of Microarchitecture Properties 1338
Pipelining, Out-of-Order, and Speculative Executions 1338
Reasoning About Memory Hierarchy 1342
Verification of Execution Units 1343
Deep Dive: Formalization and Analysis of (Simplified) x86 1345
Approach 1345
Application: Verifying x86 Instruction Implementations 1350

The work presented in this chapter was done when this author was at Centaur Technology, Inc., and prior to that, The University of Texas at Austin..

S. Goel
Intel Corporation, Austin, TX, USA
e-mail: shilpi.goel@intel.com

S. Ray (✉)
Department of ECE, University of Florida, Gainesville, FL, USA
e-mail: sandip@ece.ufl.edu

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_38

Theorem Proving Beyond Microarchitecture 1358
Conclusion 1358
References 1359

Abstract

Theorem proving is a technology where we use logical deduction to prove properties of mathematical artifacts, often assisted by a computer program called a *theorem prover*. One way to verify computing systems is to model them as mathematical artifacts and then use theorem proving to prove their desired properties as theorems. This approach has in fact been used to verify a wide spectrum of properties of computing systems. In this chapter, we recount the role of theorem proving in microprocessor verification and discuss the scope, applicability, and limits of the technology.

Keywords

Theorem proving · ACL2 · Verification · Microarchitecture · Instruction-set architecture

Introduction

Microprocessor systems are ubiquitous in today's world. They control medical monitoring equipment, banking, traffic control and transportation, and many other operations. Many of these systems are *safety critical*, and the failure of a system might cause catastrophic loss of money, time, and even human life. It is crucial for our well-being to ensure that computing systems behave correctly and reliably.

Unfortunately, ensuring reliable behavior of a modern microprocessor is a challenging problem. A microprocessor today has computing power several times that of a large supercomputer of thirty years ago. Implementations of such systems typically involve megabytes of code; even a description of their desired properties, if written down, can be hundreds of pages long. Unsurprisingly, such implementations often contain subtle bugs that are difficult to detect. The currently practiced methods for ensuring reliable executions of most system designs principally involve extensive simulation and testing. However, essential as they are, they are now proving inadequate due to the computational demands of the task. For example, it is impossible to simulate in any reasonable time the execution of even a floating-point unit in a modern microprocessor on a substantial fraction of possible inputs. Furthermore, simulation and testing are usually designed to detect only certain well-defined types of errors. They can easily miss a subtle design fault that may cause unexpected trouble only under a particular (rarely activated) set of conditions.

Formal verification has emerged as an attractive approach for ensuring correctness of computing systems. In this approach, one models the system in a mathematical logic and formally proves that it satisfies its desired specifications. Formal verification in practice makes use of some mechanical reasoning tool, i.e.,

a trusted computer program which is responsible for guiding the proof process and checking the validity of the constructed proof. When successful, the approach provides a high assurance in the reliability of the system, *viz.*, a mathematical guarantee of its correctness up to the accuracy of the model and the soundness of the computer program employed in the reasoning process. The approach is particularly enticing since, unlike simulation and testing, the guarantee is provided for *all* system executions.

This chapter focuses on microprocessor verification research through a quintessential formal verification technique: *mechanical theorem proving*. In this approach, the focus is to formalize and prove – with the assistance of a computer program referred to as the *theorem prover* – properties of computing systems just like one would prove any other mathematical formula, using standard mathematical techniques like induction, term rewriting, generalization, etc. Theorem provers are general-purpose tools which have been used to prove mathematical results as well (Shankar 1997; Russinoff 1992; Paulson 1993, 1995). Typically, no specialized mathematical foundation is provided specifically for microprocessor proofs *per se*, although certain proof techniques might be better suited for verifying certain properties. One consequence of the generality is that theorem proving is not automatic in general, and its successful use for proving nontrivial theorems about complicated systems depends on significant interaction with a trained user. The user must be familiar with the formal logic of the theorem prover as well as the nuances of the system being verified. Interacting with a theorem prover "feels" like constructing a *very* careful mathematical argument for its correctness, with the prover checking the correctness of the low-level details of the argument. Contrast this with other so-called "automated" formal verification approaches like *model checking*, *equivalence checking*, or *assertion-based verification*, where the logic and formalism used is specifically tailored for capturing properties of computing systems; verification reduces to an algorithm to check if the system (modeled using that formalism) satisfies the property.

Obviously, all other things remaining equal, employing an automated verification tool is preferable to a framework that requires close user involvement of a trained user. Why then would one want to use theorem proving rather than the more automated approaches? The key reason is that other things are *not* equal. In particular, automated algorithms incur high computational complexity. In practice, they can only be fully automatic for very small systems; for larger systems, these approaches are limited by the available time or memory. Furthermore, most theorem provers afford a substantial degree of control in the process of derivation of complex theorems. This can be exploited by the user in different forms, typically by proving key intermediate lemmas that assist the theorem prover in its proof search. By manually structuring and decomposing the verification problem, the user can guide the theorem prover into proofs about very complex systems.

There are several theorem provers available today, including ACL2 (Kaufmann et al. 2000b), Coq (Dowek et al. 1991), Forte (Aagard et al. 2000), HOL (Gordon and Melham 1993), Isabelle (Nipkow et al. 2002), and PVS (Owre et al. 1992). The underlying logics of theorem provers vary considerably, spanning across set

theory, constructive type theory, first-order logic, higher-order logic, etc. There is also substantial difference in the amount of automation provided by the different theorem provers; some are proof checkers, while others can do a considerable amount of unassisted reasoning. Different theorem provers have been employed for reasoning about a variety of architectural, microarchitectural, and hardware features of microprocessors. It is impossible to provide a thorough account of all this work in a single chapter. Rather this chapter reviews some key highlights of this extensive research. We will discuss how different features are formalized, provide a flavor of the modeling and reasoning involved, and recount some of the success stories in application of theorem proving in this area on a large scale. We will use the ACL2 theorem prover (Kaufmann et al. 2000a,b) for illustrating many of the reasoning and concepts involved. ACL2 has been used extensively for verification of a variety of microarchitectures, ranging from different (simplified) variants of x86 and JVM bytecode. However, we use ACL2 only for demonstration purposes. No prior knowledge of the theorem prover is assumed, and many of the approaches discussed are applicable to other theorem provers as well. A basic overview of ACL2 features relevant to the discussion will be provided in Section "ACL2 Preliminaries".

The remainder of the chapter is organized as follows. Section "ACL2 Preliminaries" presents the relevant background on theorem proving, including the overview of ACL2 mentioned above. Section "ISA Analysis" presents approaches to formalize the Instruction Set Architectures (ISAs) and recounts application of theorem proving at this architectural level. Section "Analysis of Microarchitecture Properties" provides a similar overview of verification of microarchitecture features, including pipelining, cache coherence, and execution units. In Section "Deep Dive: Formalization and Analysis of (Simplified) x86", we dive deeper into one specific formalization work, *viz.*, the use of ACL2 in reasoning about x86. Section "Theorem Proving Beyond Microarchitecture" goes a bit beyond architecture in two ways: the use of theorem proving to verify hardware implementation of specific microarchitecture components on the one side and software binaries on the other. We conclude in Section "Conclusion".

ACL2 Preliminaries

An obvious prerequisite for applying theorem proving for verification of a computing system is the ability to formalize the target system (and its desired properties) in the logic of a theorem prover. This, of course, presupposes that the person doing such modeling understands something about that target logic. In this section, we will provide a brief description of the logic of one theorem prover ACL2. The goal of this chapter is not for the reader to really understand logical foundations: in fact, for most of the work on actually using a theorem prover to model and reason about a system, users of a theorem prover can ignore the underlying foundations. Nevertheless, we provide a quick background of the theorem prover here for two reasons. First, when we discuss a concept informally in the subsequent section (e.g., the definition of

a machine execution via functions such as *step* and *run*), we should understand what it means for doing so in a theorem prover and how it is reconciled with the logic. Second, theorems that correspond to properties of large computing systems are themselves complex. Many of the theorems can take megabytes even to state! Enabling mathematical proofs at this scale requires a variety of automated reasoning features, and it is important that the reader understands the flavor of the features and scale of engineering involved in practical theorem proving systems. All that said, the reader does not need to understand everything about the underlying system to follow the discussions in the next sections. We encourage the reader to skim this section to get a general idea and come back to it later as a reference when they want to understand how the models and theorems presented in later sections are really formalized in the logic.

Logic Basics

A formal logic consists of (1) a *formal language* for describing formulas, a set of formulas called *axioms*, and a set of *inference rules* that allow derivation of new formulas from old ones. The key idea is to interpret the axioms as (self-evident) truths for the artifact (or *universe*) being modeled by the logic and the inference rules as validity preserving; consequently, any formula (referred to as *theorem*) derived from axioms by applying a sequence of inference rules will also be true. We will use the logic of the ACL2 theorem prover for most formulas shown in this chapter. This logic is a quantifier-free first-order logic of recursive functions with equality. The kernel of the ACL2 logic (Kaufmann and Moore 1997) consists of a formal syntax, axioms, and some rules of inference. The kernel syntax describes terms composed of variables, constants, and function symbols applied to a fixed number of argument terms. The kernel logic introduces the notion of "formulas" as composed of equalities between terms and the usual propositional connectives. The logic supported by the theorem prover is an extension of the kernel logic as described below.

The syntax of ACL2 is the prefix-normal syntax of Lisp (CLHS (Common Lisp HyperSpec)): the application of a binary function f on arguments a and b is represented by $(f\ a\ b)$ rather than the more traditional $f(a, b)$. However, in this chapter, we typically use the latter form, referring to the formal syntax only when it is relevant for the discussion. We also use more conventional notations for commonly used functions, thus writing $(x \times y)$ instead of `(* x y)` and (*if* x *then* y *else* z) instead of `(if x y z)`, dropping parentheses when it is unambiguous to do so.

ACL2 has axioms specifying properties of certain Common Lisp primitives. We show below the axioms about the primitives *equal* and *if*. Note that the kernel syntax is quantifier-free, and each formula is implicitly universally quantified over all free variables in the formula.

Axioms.

$$\begin{aligned} x = y &\Rightarrow \quad \mathit{equal}(x, y) = \mathtt{T} \\ x \neq y &\Rightarrow \quad \mathit{equal}(x, y) = \mathtt{NIL} \\ x = \mathtt{NIL} &\Rightarrow (\mathit{if}\ x\ \mathit{then}\ y\ \mathit{else}\ z) = z \\ x \neq \mathtt{NIL} &\Rightarrow (\mathit{if}\ x\ \mathit{then}\ y\ \mathit{else}\ z) = y \end{aligned}$$

The axiomatization of *equal* and *if* makes it possible to embed propositional calculus and equality into the term language. Indeed, an ACL2 user always writes terms, never formulas. Terms are interpreted as formulas by using the following convention. When we write a term τ where a formula is expected, it represents the formula $\tau \neq \mathtt{NIL}$. Thus, in ACL2, the following term is an axiom relating the Lisp functions *cons*, *car*, and *equal*:

Axiom.

$$\mathit{equal}(\mathit{car}(\mathit{cons}(x, y)), x)$$

The axiom stands for the formula $\mathit{equal}(\mathit{car}(\mathit{cons}(x, y)), x) \neq \mathtt{NIL}$, which is provably equal to $\mathit{car}(\mathit{cons}(x, y)) = x$. In this chapter, we will feel free to interchange terms and formulas by the above convention. We will also apply the same logical connectives to a term or formula; thus, when we write $\neg\tau$ for a term τ, we mean the term (or formula) $\mathit{not}(\tau)$, where *not* is axiomatized as follows.

Axiom.

$$\mathit{not}(x) = \mathit{if}\ x\ \mathit{then}\ \mathtt{NIL}\ \mathit{else}\ \mathtt{T}$$

The duality between terms and formulas enables us to interpret an ACL2 theorem as follows. If the term τ (interpreted as a formula) is a theorem, then for all substitutions σ of free variables in τ to objects in the ACL2 universe, the (ground) term τ/σ evaluates to a non-NIL value; NIL can thus be viewed as logical false.

The kernel logic includes axioms that characterize the primitive Lisp functions over numbers, characters, strings, constant symbols such as T and NIL, and ordered pairs. These objects together make up the ACL2 standard universe, but the axioms do not preclude "nonstandard" universes which may contain other objects. Lists are represented as ordered pairs, so that the list `(1 2 3)` is represented by the term $\mathit{cons}(1, \mathit{cons}(2, \mathit{cons}(3, \mathtt{NIL})))$. For brevity, we will write $\mathit{list}(x, y, z)$ as an abbreviation for $\mathit{cons}(x, \mathit{cons}(y, \mathit{cons}(z, \mathtt{NIL})))$. Another convenient data structure built out of ordered pairs is the *association list* (or *alist*) which is essentially a list of pairs, e.g., $\mathit{list}(\mathit{cons}(\text{``a''}, 1), \mathit{cons}(\text{``b''}, 2))$. We often use alists for describing finite mappings; the above alist can be thought as a mapping that associates the strings `"a"` and `"b"` with 1 and 2, respectively.

In addition to propositional calculus and equality, the rules of inference include instantiation and well-founded induction up to ε_0 (Here ε_0 is the least ordinal

that is closed under exponentiation. We do not discuss ordinals here. For a comprehensive overview of ordinals, we refer the reader to Church and Kleene's treatment (Church and Kleene 1937). For this paper it is sufficient to think about ordinals as extension of natural numbers. For most of the theorems we prove for computing systems we need to only think about natural numbers or a lexicographic tuple of natural numbers.). For instance, the formula $\mathit{car}(\mathit{cons}(2, x)) = 2$ is provable by instantiation from the above axiom relating *car*, *cons*, and *equal*. The ACL2 theorem prover initializes with a boot-strapping first-order theory called the *Ground-Zero* Theory (GZ for short), which contains the axioms of the kernel logic. The logical foundation for induction in ACL2 is provided as follows. GZ contains an embedding of ordinals up to ε_0, represented in Cantor normal form (Manolios and Vroon 2003), and GZ axiomatizes a binary relation $\prec$ to be an irreflexive total order on (the representation of) ordinals. The theory GZ is *inductively complete*: for any formula φ expressible in GZ, every first-order *induction axiom* of the following form belongs to GZ, where φ/σ denotes the formula obtained by applying the substitution σ to φ.

$$(\forall y \prec \varepsilon_0)[((\forall x \prec y)\varphi/\{y := x\}) \Rightarrow \varphi(y)] \Rightarrow (\forall y \prec \varepsilon_0)\varphi(y)$$

The formula may appear a bit complicated at first glance and can be skipped on a casual read. For the reader interested in understanding what it stands for, it can be interpreted as follows. Let y be an ordinal less than ε_0 and consider a formula $\varphi(y)$. Suppose we can prove that if $\varphi(x)$ holds for each $x \prec y$, then we can also prove $\varphi(y)$. That is, by assuming all smaller instances (according to the relation "$\prec$") of the formula $\varphi(x)$, we can prove $\varphi(y)$. Then the formula $\varphi(y)$ holds. Note that this is the crux of traditional induction, except that the concepts are adapted to induction on ordinals up to ε_0 (rather than natural numbers). ACL2 implicitly assumes all such formulas as axioms.

Finally, ACL2 only allows construction of theories that are extensions of GZ via the *extension principles* explained below, which allow axiomatization of new function symbols. When a new function symbol is introduced via the extension principles, the resulting theory $\mathscr{T}'$ is the extension of the original theory $\mathscr{T}$ with (i) the axiom explicitly introduced by the extension principle and (ii) all the induction axioms in the language of the new theory.

Extension Principles

ACL2 provides *extension principles* allowing the user to introduce new function symbols. Below, we discuss one extension principle which is particularly relevant to us, i.e., the *definitional principle* for introducing totally defined functions. Other extension principles include an *encapsulation principle* for introducing partially defined or constrained functions, a *defchoose principle* for introducing Skolem (choice) functions, and a *defaxiom principle* that enables the specification of a formula as an axiom. The latter is discouraged since the introduction of arbitrary

axioms is potentially unsound. For this chapter, unless explicitly mentioned otherwise, we ignore introduction of arbitrary axioms.

The *definitional principle* allows the user to extend a theory by axiomatizing new total (recursive) functions. For example, one can use this principle to introduce the unary function symbol *fact* axiomatized as follows, which returns the factorial of its argument.

Definitional Axiom.

$$\mathit{fact}(n) = \mathit{if}\ \mathit{zp}(x)\ \mathit{then}\ 1\ \mathit{else}\ n \times \mathit{fact}(n-1)$$

Here, $\mathit{zp}(n)$ is axiomatized in GZ to return `NIL` if n is a natural number greater than 0, and `T` otherwise. To ensure that the extended theory is consistent, ACL2 first proves that the recursion terminates. This is done by exhibiting a *measure* that maps the list of function arguments to the set of ordinals below ε_0, and showing that the measure decreases at every recursive call. For *fact* above, one possible measure is $\mathit{nfix}(n)$ (axiomatized in GZ) which returns n if n is a natural number, otherwise 0.

The Theorem Prover

ACL2 is an automated, interactive proof assistant. It is *automated* in the sense that no user input is expected once it has embarked on the search for the proof of a conjecture. It is *interactive* in the sense that the search is significantly affected by the previously proven lemmas in its database at the beginning of a proof attempt; the user essentially programs the theorem prover by stating lemmas for it to prove, to be used automatically in subsequent proofs. ACL2 also supports a goal-directed interactive loop called the "*proof-checker*," similar in nature to LCF-style provers like HOL (Gordon and Melham 1993) and Isabelle (Nipkow et al. 2002), but it is much less frequently used and not relevant to the discussions in this chapter.

Interaction with the ACL2 theorem prover principally proceeds as follows. The user creates a theory (extending GZ) using the extension principles to model some artifact of interest. Then, she poses a conjecture about the functions in the theory and instructs the theorem prover to prove the conjecture. For instance, if the artifact is the factorial function above, one conjecture might be the following formula, which says that *fact* always returns a natural number. The function *natp* below is also axiomatized in GZ.

Theorem (`fact-is-natp`).

$$\mathit{natp}(\mathit{fact}(x)) = \mathtt{T}$$

ACL2 attempts to prove a conjecture by applying a sequence of transformations to it, replacing each goal (initially, the conjecture) with a list of subgoals. Internally, ACL2 stores each goal as a *clause* represented as an object in the ACL2 universe. A goal of the form $\tau_1 \wedge \tau_2 \wedge \ldots \wedge \tau_n \Rightarrow \tau$ is represented as a list of terms

$(\neg\tau_1 \ldots \neg\tau_n\ \tau)$, which is viewed as the disjunction of its elements (*literals*). ACL2 has a *hint* mechanism which the user can use to provide pragmatic advice on proof search at any goal or subgoal; in this example, the user can advise ACL2 to begin the search by inducting on `x`. Once a theorem is proven, it is stored in a database and used in subsequent derivations. This database groups theorems into various *rule classes*, which affects how the theorem prover will automatically apply them. The default rule class is *rewrite*, which causes the theorem prover to replace instances of the left-hand side of an equality with its corresponding right-hand side; if the theorem `fact-is-natp` above is stored as a rewrite rule, then if ACL2 subsequently encounters a term of the form *natp*(*fact*(τ)), then the term is rewritten to `T`.

Some Execution Features: Guards, `MBE`, and `Stobjs`

ACL2 is closely tied with Common Lisp. It employs Lisp syntax, and as of this writing, ACL2 can be built on top of all major Common Lisp distributions (Allegro Common Lisp, CCL, GCL, LispWorks, and SBCL). Furthermore, events corresponding to the definitional principle are Lisp definitions. For instance, the formal event introducing *fact* also serves as a Common Lisp definition:

```
(defun fact (n)
  (if (zp n)
      1
    (* n (fact (- n 1)))))
```

The connection with Lisp enables users to execute formal definitions efficiently; ACL2 permits the execution of all functions axiomatized in GZ, as well as any function whose definition does not involve any constrained functions. The theorem prover makes use of this connection for simplifying ground terms. For instance, during a proof, ACL2 will automatically simplify *fact*(3) to 6 by evaluation (also referred to as "concrete execution").

The fact that the same functions are used for concrete execution as well as formal reasoning offers the advantage that a model of a computing system built in ACL2 can be validated against the real system or a trusted "golden" model by running co-simulations. This increases confidence in the accuracy of the formal model and, by extension, in the guarantees offered by formal analysis done using this model. For instance, one accomplishment of ACL2 is the formal verification of a formal microarchitectural model of Rockwell Collins AAMP™ processor (Greve et al. 2000). The formal proof shows that the formal model satisfies the desired property but leaves open the possibility that the physical artifact has not been accurately captured by the formalization. The latter question can be effectively answered (and was in fact addressed in this case) by extensive co-simulation of the actual artifact with the formal model. The x86 formalization we discuss in Section "Deep Dive: Formalization and Analysis of (Simplified) x86" has also been "vetted" with the real implementation through such co-simulation. Below, we discuss some features offered by ACL2 to optimize the execution efficiency of its functions.

Intended Domains and Guards

An interesting difference between ACL2 and Common Lisp is that Common Lisp functions are *partial* while functions in ACL2 are total. The Common Lisp standard (CLHS (Common Lisp HyperSpec)) specifies the *intended domain* of each Lisp primitive function; the return value of the function is defined for inputs in that intended domain. For instance, the intended domain of the Lisp primitive function $\mathit{car}(x)$ (which informally returns the first object of an ordered pair) is given by the formula $\mathit{consp}(x) \vee (x = \texttt{NIL})$, where $\mathit{consp}(x)$ is another Lisp primitive that returns `T` if x is an ordered pair, and `NIL` otherwise. Based on the formula, the number 3 is outside the intended domain of *car*. Consequently, the return value of $\mathit{car}(3)$ is undefined by the standard, and different Lisp implementations can return different values. Indeed, the same Lisp implementation can return different values for $\mathit{car}(3)$ in different invocations. However, GZ includes axioms that specify unique return values of all such primitive functions, even outside the intended domain. The axioms are crafted so that the return value of the function is consistent with the Common Lisp return value for arguments in the intended domain, while a reasonable "default" return value is provided for arguments outside the domain via additional *completion axioms* (Kaufmann and Moore 1994, 1997). Following is the axiom for *car*. Based on this axiom, we can prove the formula $\mathit{car}(3) = \texttt{NIL}$.

Completion Axiom.

$$\neg \mathit{consp}(x) \vee (x = \texttt{NIL}) \Rightarrow \mathit{car}(x) = \texttt{NIL}$$

The above implies that (1) it is unsound to simply execute a function in Lisp to determine the return value specified by the axioms on concrete inputs if the inputs are outside the intended domain, but (2) it is sound to do so if the inputs are within the intended domain. The notion of *guards* formalizes this idea of intended domain. A guard of a function is simply a formula $\mathscr{G}$ that must be satisfied to ensure that if the inputs satisfy $\mathscr{G}$, then the return value of the function is consistent with Common Lisp. For instance, the guard of $\mathsf{car}(x)$ is the formula $\mathsf{consp}(x) \vee (x == \texttt{NIL})$.

The notion of guards extends to user-defined functions as well. For example, consider defining a new function two as follows. The function always returns the number 2 (and can be proven as such by ACL2).

Definitional Axiom.

$$\mathit{two}(x) = \mathit{if}\ \mathit{eq}(\mathit{car}(x), 2)\ \mathit{then}\ \mathit{car}(x)\ \mathit{else} 2$$

In order to use Common Lisp for evaluating the function *two*, the guard of any function called involved in its body must be satisfied. Here there are two such functions, both Common Lisp primitives, *eq* and *car*. The function *eq* is simply equality but is a faster variant implemented by fast pointer check. The guard for $\mathit{eq}(x, y)$ is $\mathit{symbolp}(x) \vee \mathit{symbolp}(y)$ where *symbolp* returns `T` when its argument is a symbol and `NIL` otherwise. Putting this together with the guard of *car*, the guard of the function *two* is given by the formula $(\mathit{consp}(x) \wedge \mathit{symbolp}(\mathit{car}(x))$.

ACL2 provides a mechanism for verifying guards of a function. Any guard-verified function can be executed by the Common Lisp evaluator for arguments satisfying the guards; ACL2 can still evaluate the function otherwise, but it is much slower since the Lisp evaluator cannot be directly invoked.

Must Be Equal

ACL2 uses guards to produce other more sophisticated features supporting fast execution. One such feature is *must-be-equal* (MBE for short) (Greve et al. 2008). The key idea is to enable two definitions of a function, one to be used for logical reasons (e.g., for proofs) and another for fast execution efficiency. To motivate the idea, consider the function *lng* as follows, which computes the length of a list.

Definition.

$$lng(x) = \mathit{if}\ consp(x)\ \mathit{then}\ lng(cdr(x)) + 1\ \mathit{else}\ 0$$

Here, the function *cdr* is a primitive Lisp function that takes a list as argument and returns a modified list after removing the first element. The function *lng* can have a guard $consp(x) \vee (x = \texttt{NIL})$, and such a guard can be verified by ACL2. Even after guard verification; however, evaluation of *lng* on a large list argument with Common Lisp will typically encounter a stack overflow because of the recursive call. A more execution-friendly definition could be the following:

Definition.

$$lnga(x, a) = \mathit{if}\ consp(x)\ \mathit{then}\ lnga(cdr(x), a + 1)\mathit{else}\ a$$
$$lng(x) = lnga(x, 0)$$

Since the function *lnga* is tail recursive, good Common Lisp compilers will compile this function into a simple loop with no stack allocation on recursive function calls. On the other hand, it is a more complex function to reason about. The MBE feature enables the user to use these two different definitions of *lng* – the user uses the first definition as a logical definition and the second with a directive to use for execution purpose only.

Definition.

$$\begin{aligned} lng(x) = MBE &: \texttt{logic}\ \mathit{if}\, consp(x)\, \mathit{then}(lng(cdr(x)) + 1)\, \mathit{else} 0 \\ &: \texttt{exec}\quad lnga(x, 0) \end{aligned}$$

The guard for MBE includes the proof obligation that the two definitions are equal (under the guard of the function), a nontrivial one-time cost. This proof obligation is part of the guard verification, i.e., conceptually, one imagines the guard of MBE to be the requirement that the two definitions are logically equivalent.

Single-Threaded Objects

The third execution feature of ACL2 we cover in this chapter is *single-threaded objects* ("stobj" for short) that provides the benefit of destructive updates (Boyer and Moore 2002). Note that a fundamental tenet of mathematics is that the function $f(x)$ returns the same value for each invocation with the same argument x; i.e., $(x = y) \implies (f(x) = f(y))$. This notion is, of course, not true if f is implemented in a computer program (even if f does not have side effects) where x is a variable: x can be updated (e.g., by an assignment) between two successive invocations of $f(x)$. The idea of stobjs is to enable declaration of variables that can be destructively updated while enabling logical reasoning. Following is an ACL2 declaration that defines the variable `obj` to be a single-threaded object.

```
(defstobj obj
  (field-a :type (array (unsigned-byte 64) (1024)))
  (field-b :type (signed-byte 16)))
```

Logically, `obj` is a list of two elements; the first element `field-a` is itself a list of 1,024 elements, each of which is a 64-bit unsigned integer, and the second element `field-b` is a 16-bit signed integer. Under the hood, `obj` is implemented as a one-dimensional array with `field-a` defined as a simple array of 1,024 64-bit unsigned integers and `field-b` as another simple array containing 1 16-bit signed integer element. Furthermore, the above declaration also defines functions *update-obj*, *update-field-a*, *update-field-b*, etc. Logically, *update-field-a* returns a (new) list with the appropriate entry updated, *update-field-b* returns a new value, and *update-obj* returns a new pair. However, again under the hood, the updates are actually implemented destructively on the object `obj`. The logical view is reconciled with the implementation by enforcing syntactic restrictions on the manner in which the functions manipulating stobjs can be invoked. Roughly, every function that updates (any field of) a stobj must return the updated stobj. Furthermore, updates to different fields of stobj are sequentialized, and two functions f and g cannot take the same stobj and update different fields; instead, f (resp., g) must take the updated stobj returned by g (resp., f) to perform its own updates.

Single-threaded objects are crucial to defining efficient ISA and microarchitectural models. As we will see in the next section, these models often involve defining how different instructions update machine states. Fast simulation of these models depend on fast (destructive) updates to machine states. In the `x86isa` model, discussed in Section "Deep Dive: Formalization and Analysis of (Simplified) x86", we will see the use of stobjs to specify the x86 machine state.

ISA Analysis

The Instruction Set Architecture (ISA) is an abstraction providing a view of the hardware that is appropriate for software execution. Much of the implementation details are hidden, e.g., pipeline, cache, TLB, speculation primitives, etc. Instead,

the view provided is of a machine that executes (binary) code one instruction at a time. This section discusses ISA formalization and its applications.

ISA Formalization

An obvious approach to formalizing an ISA is to define the effect of executing an instruction on the programmer-visible components of the machine (e.g., architectural registers, memory, etc.). We can define a function *execute-inst*$(s, inst)$ as follows. Let s be a tuple of values $\langle regs, mem, \ldots \rangle$ representing the architectural state. For our purpose, the machine state is represented as a tuple of values of all machine variables (or *components*). Let *pc*(s) be the function that given a state s gives the value of the program counter, and let *mem*(s) represent the instruction memory. For instance, if *regs* is the first component of s, *mem* is the second component of s, and *pc* is the third component of *regs*, we can define the next instruction to be executed at a state s by the function *instr* as follows.

Definition.

$$\begin{aligned} &\mathit{pc}(s) = \mathit{nth}(3, \mathit{nth}(0, s)) \\ &\mathit{mem}(s) = \mathit{nth}(1, s) \\ &\mathit{instr}(s) = \mathit{nth}(\, \mathit{pc}(s), \mathit{mem}(s)) \end{aligned}$$

To formalize the notion of state transition, we first define a binary function *ISA.effect*. Given an instruction I and a state s, *ISA.effect*(s, I) returns the state s' obtained by executing the instruction I from state s. For example, if I is the instruction `LOAD`, then its effect might be to push the contents of some specific variable on the stack and increase the program counter by some specific amount. We can then define our state transition function *ISA.step* as follows, such that for any state s, *ISA.step*(s) returns the state of the machine after executing one instruction.

Definition.

$$\mathit{ISA.step}(s) = \mathit{ISA.effect}(s, \mathit{instr}(s))$$

Obviously, the representation of a machine state as a *list* of components in our description above is merely for the purpose of illustration. Different machines are formalized in different ways; for example, the states might be modeled using arrays or association lists instead of lists. In what follows, the actual formal representation of the machine states or the actual definition of *ISA.step* is mostly irrelevant. What we assume is that there is *some* formal representation of the states in the formal theory, and given a state s, *ISA.step*(s) can be interpreted to return the state of the machine after executing one instruction from s. This can always be done as long as we are concerned with reasoning about deterministic sequential programs.

It will be convenient for us to define a new function *ISA*.run as follows to return that state of the machine after n *ISA.step*s from s.

Definition.

$$\mathit{ISA.run}(s, n) = \mathit{if}\, \mathit{zp}(n)\, \mathit{then}\, s\, \mathit{else} \mathit{ISA.run}(\mathit{ISA.step}(s), n - 1)$$

Mechanical Analysis for ISA

The function *ISA.run* computes the architectural state of the machine after executing n instructions. The function *ISA.step* (and correspondingly, *ISA.run*) formalize the ISA, but they are mathematical functions about which one can prove different properties. Here is one such "obvious" property about the *ISA.run* function, which is easy to prove by induction on m. Note that the property is true for *ISA.run* independent of the definition of *ISA.step*.

Lemma.

$$\mathit{natp}(m) \wedge \mathit{natp}(n) \Rightarrow \mathit{ISA.run}(s, m + n) = \mathit{ISA.run}(\mathit{ISA.run}(s, m), n)$$

The above lemma is illustrative and fundamental. But it is not perhaps "interesting" to an architect looking to verify properties of a specific ISA. However, one can verify ISA-specific properties as well. One interesting direction is to relate two different ISA models. In particular, consider designing an abstract and highly simplified ISA model specified by the function *ISA.abstract.step* that defines arithmetic operations as mathematical function. For instance, we define the corresponding effect function *ISA.abstract.effect* such that if the instruction I is an `ADD`, then the result that is stored in the target register is the infinitely precise mathematical sum of the operand. Obviously, such an ISA would not be realizable in any practical architecture. But it is a useful mathematical abstraction, since after all we think about the `ADD` operation as some approximation of the mathematical sum. Once this is done, we can, of course, formalize another more elaborate ISA model *ISA.practical.step* in which the arithmetic operations are formalized to include all practical bells and whistles, e.g., rounding, truncating, overflow and underflow flags, etc. We can then prove a theorem roughly stated as follows:

> If a program never raises any overflow or underflow errors, then the execution of the program on the simplified ISA model has the same effect as executing it under the more elaborate model.

Obviously, one instance of the elaborate model is the corresponding microarchitecture models. The connection between microarchitecture and ISA is a specific and active area of research, and we consider that exclusively in Section "Analysis of Microarchitecture Properties". However, other elaborations have been performed as well. For instance, at the University of Texas, ACL2 models have been developed with increasing levels of elaboration for the Java Virtual Machine, starting with a simple "toy" model called M1 to a highly realistic model M6 that can execute real JVM bytecode with high fidelity (Moore 2003). Obviously, we do not have space here to discuss all the formalizations; a quick summary will be provided in

Section "Some Formalized ISAs". But we bring that up here to illustrate a specific proof that was done to show correspondence between two models M3 and M4. The key difference between the two models is multi-threading: M4 is a multi-threaded model while M3 is not. A theorem that was formalized and mechanically proven by ACL2 relating these two machines can be roughly paraphrased as follows.

> Let π be any M4 program that never spawns a thread. Then, the effect of executing π on M4 is the same as the effect of executing π on M3.

Of course, care is necessary to make the statement precise. Formalizing the statement "the effect is the same in both machines" entails designing a function (referred to as the *projection function*) that takes an M4 state s, eliminates components that are irrelevant to multithreading (e.g., registers keeping track of different threads) and creates a state s'that can be used by the state transition function for M3.

Binary Code Analysis with ISA Models

An obvious application of ISA formalization is verification of binary (or assembly) code: we prove that if the machine executes from a state s satisfying the program's precondition, then the state reached on completing the execution satisfies some desired postcondition. Here, the state s is the architecturally visible state. The ISA model is used to assign meaning to the program. To understand how this works, consider the program in Fig. 1. The program consists of two variables `X` and `Y` and simply loops 10 times incrementing `X` in each iteration. In the figure, the number to the left of each instruction is the program counter value for the loaded program. The postcondition we might prove is that the variable `X` holds the value `10`, and the precondition can specify that the program binary is loaded in the memory from the address (purportedly location 1) and the program counter $\mathit{pc}(s)$ points to the memory location. Assume that *prog-loaded* is a predicate that holds for a state s if and only if the program shown has been loaded in s starting from location 1. The precondition and postcondition above can be defined as follows.

Definition.

$$\mathit{pre}(s) = (\mathit{pc}(s) = 1) \wedge \mathit{prog\text{-}loaded}(s)$$
$$\mathit{post}(s) = (X(s) = 10)$$

```
1:  X:=0;                          {T}
2:  Y:=10;
3:  if (Y ≤ 0) goto 7;             {(X+Y) = 10}
4:  X:=X+1;
5:  Y:=Y-1;
6:  goto 3;
7:  HALT                           {X = 10}
```

Fig. 1 A simple one-loop program

Of course, defining the precondition and postcondition only amount to *specifying* the correctness of the program. Carrying out the proof can actually be nontrivial. There are many classic approaches to program verification (Goldstein and von Neumann 1961; Floyd 1967). Here, we show one approach, called *clock functions*. Note that from a logical perspective, all the common approaches are known to have the same logical strength, i.e., a proof in one approach can be mechanically converted to a proof in another (Ray and Moore 2004; Ray et al. 2008). The clock functions approach entails defining a function $\mathit{clock}(s)$ that maps every state s to a natural number that specifies the number of steps required to terminate from s. We then prove the following formula as a theorem:

Total Correctness Theorem.

$$\mathit{pre}(s) \Rightarrow \mathit{post}(\mathit{ISA.run}(s, \mathit{clock}(s)))$$

The clock function for this program can be defined as follows. Note that the definition closely mimics the actual loop structure of the program. In particular, consider the function lpc (which stands for "loop clock"). If the machine is in a state s where the program counter has the value 3, then $\mathit{lpc}(s)$ merely counts the number of steps before the loop is exited.

Definition.

$$\mathit{lpc}(s) = \mathit{if}\ \mathit{zp}(Y(s)) \vee \neg \mathit{prog\text{-}loaded}(s)\ \mathit{then}\ 0\ \mathit{else}\ 4 + \mathit{lpc}(\mathit{ISA.run}(s, 4))$$
$$\mathit{clock}(s) = 2 + \mathit{lpc}(\mathit{ISA.run}(s, 2)) + 1$$

Once the appropriate clock function is defined, we can try to prove the total correctness theorem. To do so, we must prove a theorem that characterizes the loop itself. One possibility is to prove the following lemma:

Lemma.

$$\mathit{natp}(Y(s)) \wedge \mathit{prog\text{-}loaded}(s) \wedge (\mathit{pc}(s) = 3) \Rightarrow$$
$$\mathit{ISA.run}(s, \mathit{lpc}(s)) = \mathit{upd}(s, X(s) + \mathit{lpc}(s), Y(s) - \mathit{lpc}(s))$$

Here, $\mathit{upd}(s, a, b)$ is the state obtained by assigning the value a to component `X` and the value b to component `Y`, respectively, in state s. The formula can be proven as a theorem by induction based on the term $\mathit{lpc}(s)$. The proof of the theorem

Total Correctness then follows from this theorem and the lemma on the composition of $\mathit{ISA.run}$.

Some Formalized ISAs

Formal ISA models have long been used as a specification for the verification of processor microarchitectures as well as machine code. Hunt's FM8501 processor (Hunt

1994) was an early case study which successfully expressed a generic processor's specification and design in the formal logic of a theorem prover. The CLI stack project (Bevier et al. 1989) used NQTHM (Boyer et al. 1995), a predecessor of the ACL2 theorem prover, to formally verify a "stack" of systems, from a gate-level microprocessor design (Hunt 1989) to an assembler (Moore 1996) that compiled to this microprocessor and, finally, a higher-level language (Young 1989) that targeted this assembler. The CLI stack set a milestone in the history of the design and verification of software and hardware and continues to inspire many undertakings like the recent Provably Correct Systems (ProCoS) projects (He et al. 1994). Sawada et al. used ACL2 to verify the pipelined execution of FM9801 (Sawada and Hunt 2002b), a processor that supported speculative execution, out-of-order instruction issue and completion, and exceptions and interrupts; the correctness property was that FM9801's pipelined execution from one flushed state to another is comparable to sequential execution. Rockwell Collins developed a formal model (Greve 1998) of their JEM1 microprocessor in the PVS theorem proving system (Owre et al. 1992) for microcode verification. A Rockwell Collins project used ACL2 to verify that the microcode of AAMP7G™ processor was compliant with the EAL 7 standard security specification (Wilding et al. 2010).

Boyer and Yu formalized most of the user-mode instruction set of the Motorola MC68020 microprocessor (Boyer and Yu 1996) in NQTHM. This ISA model was used to verify the machine code corresponding to the Berkeley string library. Lie and Moore formalized the Java Virtual Machine (JVM) in ACL2 (Liu and Moore 2004) in order to reason about JVM bytecode. A subset of the x86 ISA was formalized in ACL2 and used to perform machine-code verification of both system and application programs. This model was later used to verify parts of the x86 microarchitecture; details are in Section "Deep Dive: Formalization and Analysis of (Simplified) x86". Another specification of the x86 ISA (Degenbaev 2012) formalized the x86 instruction semantics using a domain-specific language and specified the total-store-ordered memory model by accounting for caches, translation look-aside buffers, fences, locks, etc.

The CHERI (Capability Hardware-Enhanced RISC Instructions) ISA (Watson et al. 2016) is an architecture that provides software compartmentalization by supporting a hybrid capability model (Levy 1984) at the level of the processor itself. Formal models of CHERI ISA have been developed using L3 (Fox 2015), a domain-specific language for instruction-set descriptions, and PVS. Recently, Arm released executable specifications of its ISA (Arm ISA Specifications), where instructions' semantics have been formalized using a domain-specific specification language called ASL. Arm has successfully used these specifications to verify parts of the Arm microarchitecture (e.g., pipeline control logic) (Reid 2016; Reid et al. 2016). This ASL specification has also been translated into Sail (Armstrong et al. 2019), an open-source domain-specific language intended for specifying ISAs, and from Sail to the Isabelle/HOL theorem prover (Nipkow et al. 2002; Gordon and Melham 1993); the resulting Isabelle/HOL definition was used to reason about the compartmentalization (CHERI) properties of the Arm-based Morello architecture (Bauereiss et al. 2021).

Analysis of Microarchitecture Properties

The ISA formalization discussed in Section "ISA Formalization" shows the basics of how one can formalize a computing system, e.g., by defining the state transition function in the logic of the theorem prover. The high-level modeling approach applies essentially unchanged to modeling a microarchitecture, except that the "effect function" is more detailed. Just like the function $\mathit{ISA.effect}(s, I)$ models the effect of executing instruction I on the *architectural* state s, one can define a function $\mathit{ma\text{-}effect}(m, I)$ that defines the effect of "executing" I on a microarchitectural state m. Note that we have put the term "executing" in quotes: in a microarchitecture, the notion of executing varies depending on the microarchitectural component being investigated. For instance, if we are analyzing an in-order pipelined microprocessor, $\mathit{ma\text{-}effect}(m, I)$ may define the effect on the microarchitecture state as the instruction moves from one pipeline stage to another. On the other hand, for an out-of-order processor, $\mathit{ma\text{-}effect}(m, I)$ would need to account for transition of the instruction through the issue queue, reservation station, reorder buffer, etc.

Pipelining, Out-of-Order, and Speculative Executions

One key microarchitectural optimization is pipelining. Pipelining enables overlapped execution of different instructions so that an instruction can be initiated before its preceding instruction is completed. The basic pipelining idea has, of course, been extended and augmented with various advanced features such as out-of-order executions, speculative executions, etc. Unfortunately, these features significantly complicate verification of microprocessors. One obvious reason for this is just the additional complexity: proofs of complex artifacts are complex. However, the complexity in this case manifests itself not just in the complexity of the proof but also in the complexity of the specification. In other words, it is sometimes unclear *what should be proven* to infer that the microarchitectural implementation is indeed correct. Over the last three decades, there has been numerous notions of correctness for microprocessors, and the topic has become difficult, highly technical, and even controversial (Aagaard et al. 2001). In this section, we will give a flavor of the issues involved.

Pipelining

To understand how pipelining can complicate verification, let us quickly summarize how one can verify a *non-pipelined* microarchitecture. Simulation correspondence can be informally described as follows. One defines a predicate sim as a relation between the states of MA and ISA with the following properties: (In practice, we use simulation for non-deterministic systems, which can be formalized with step taking an additional argument that models a non-deterministic input. In this chapter, we ignore issues involving non-determininstic concurrent systems.)

- $sim(MA.\ init(), ISA.init())$
- If *ma*and *isa*are two states such that $sim(ma, isa)$ holds, then

  1. $MA.\ label(ma) = ISA.label(isa)$
  2. $sim(MA.\ step(ma), ISA.step(isa))$ holds.

The predicate *sim* is referred to as a *simulation relation*. The two conditions above imply that for every execution of *MA* is matched by *ISA* such that they have the same labels. The function *label* formalizes the observable components in a state: we can think of $label(ma)$ to be the architectural components of *ma*. In microprocessor verification, one often uses a specific simulation relation that is based on so-called projection functions. In this approach, one defines a function *proj* such that, given a *MA* state *ma*, $proj(ma)$ returns the corresponding *ISA* state. Using the terminology we introduced in the previous section, we can think of *proj* as the *representative function* *rep*. The function is called *projection* since it "projects" the programmer-visible components of an *MA* state to the *ISA*. The theorem proved to show the correctness of *MA* can be stated as follows.

Correctness Theorem.

$$proj(MA.\ step(ma)) = ISA.\ step(proj(ma))$$

The theorem is shown more often as a diagram such as in Fig. 2. This can be cast as a simulation proof by defining the simulation relation to be defined as follows:

Definition.

$$sim(ma, isa) = (\ proj(ma) = isa)$$

Unfortunately, for pipelined microarchitectures, however, one cannot use simple simulation correspondence. Why? In a pipelined *MA*, when one instruction completes execution, others have already been partially executed. Thus, it is difficult to come up with a simulation relation such that the properties 1 and 2 above hold. This problem is often referred to as the *latency problem* (Srivas and Bickford 1990; Bronstein and Talcott 1990).

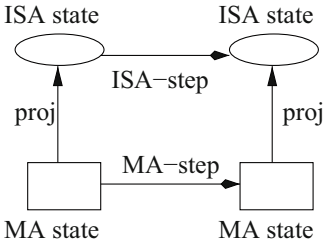


Fig. 2 Pictorial representation of simulation proofs using projection

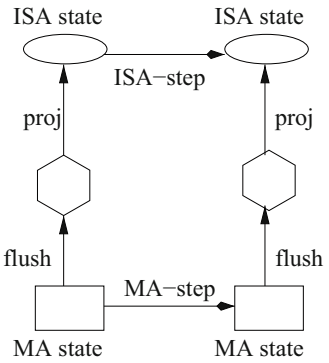


Fig. 3 Pictorial representation of flushing proofs

Flushing Proofs. One approach to address the above issue was presented by Burch and Dill in 1994 (Burch and Dill 1994), as an approach to compare *MA* states with *ISA* states, where the *MA* now is a pipelined machine. This approach is known as *flushing correspondence*. The notion is shown pictorially in Fig. 3. To construct an *ISA* state from an *MA* state, we simply flush the pipeline; that is, complete all partially executed executions in the pipeline without introducing any new instruction. We then project the programmer-visible components of this flushed state to create the *ISA* state. Then, the notion of correctness says that *MA* is correct with respect to the *ISA*, if whenever flushing and projecting an *MA* state *ma*yields the state *isa*, it must be the case that for every possible next state ma' in *MA* from *ma*, there must be a state isa' in *ISA* such that (1) isa' can be reached from *isa*in one *ISA* step, and (2) flushing and projecting from ma' yields isa'.

Unfortunately, flushing has its own limitation. First, note that the diagram shown in Fig. 3, unlike the diagram in Fig. 2, does not render itself directly to a proof of simulation correspondence or trace containment. This is because $proj(flush(s))$ maps a state s of a pipelined machine *MA* to a state with possible different values of observable components (or label). Manolios' (Manolios and Johnson 2000), certain types of flushing diagrams are flawed in that trivial, obviously incorrect machines satisfy such notion of correctness.

Interrupts, Out-of-Order and Speculative Execution, Self-Modifying Code, and the Works

The discussion on pipelining above was intended to simply give a taste of complexities associated with advanced microarchitectural control features. Note that it becomes even more difficult to use flushing proofs for advanced microarchitectural features. Consider flushing a microarchitectural state *ma*in which an interrupt has been partially serviced. Flushing would cause the system to complete all incomplete instructions at state *ma*, perhaps including an instruction i speculatively fetched after the interrupt has been detected (but not serviced). This would result in *ma* getting mapped to an inconsistent state, which has no correspondence with the actual execution of *MA*. Consequently, a notion of correctness that is based on flushing

cannot directly handle interrupts. This problem (i.e., microarchitectural states being mapped by flushing to potentially inconsistent ISA states) is exacerbated with out-of-order and speculative executions, where obviously an MA state can include partially completed executions that may be completed in a different order than if the system were flushed (out-of-order execution), or maybe even not completed at all (speculative execution).

In spite of the challenges above that have so far precluded a general notion of correctness for microprocessors with advanced control, there has certainly been significant work on formal verification of microarchitecture models with such features. Much of the work has been custom, i.e., specific correctness properties proven for a specific system. One of the most comprehensive formal proofs was from Sawada and Hunt (Sawada and Hunt 2002a), which formalized and verified microprocessor named FM9801 with features such as out-of-order and speculative execution, interrupts, exceptions, and self-modifying code. The correctness proof they verified is roughly summarized as follows:

> Let MA_0 be a flushed state and suppose after n transitions the microarchitecture reaches another flushed state MA_n. Let the projection of MA_0 be ISA_0 and let the projection of MA_n be ISA_m. Suppose the interrupt register specifies a sequence of interrupts Σ. Suppose also that there is no exception in the transition sequence of MA, and no self-modifying code is executed. Then there is a sequence of transitions from ISA_0 to ISA_m and the sequence of interrupts serviced in the process by ISA is Σ.

Of course, we note that the above theorem does suffer from the problems pointed to by Manolios: a trivial machine that never reaches the flushed state satisfies these criteria. However, the FM9801 machine is not a trivial microarchitecture, and the theorem is a nontrivial result for a microarchitecture with advanced features.

We primarily focused in this section on the notion of correctness, since that is controversial and a target of lively debate. Of course, another issue is to actually do the proof. Without standardization, these proofs tend to be highly custom-built. Nevertheless, some interesting insights have emerged over the years on how to approach the verification. One critical insight is that we must define an *invariant*, i.e., a predicate that holds for each microarchitectural state encountered during execution. In other words, one defines a function *inv* to satisfy the following conditions:

Invariant Properties.

$$\begin{aligned} &\mathit{inv}(\mathit{MA}.init()) = T \\ &\mathit{inv}(s) \Rightarrow \mathit{inv}(\mathit{MA}.\mathit{step}(s)) \end{aligned}$$

Sawada and Hunt's work showed how to do this manually. They defined an auxiliary structure, called MAETT, that is used for keeping track of the history of execution of the different instructions through the pipeline. In other words, MAETT keeps track of which instruction is at what stage of completion and how the instruction would be expected to proceed in subsequent transitions. Then *inv*

simply establishes that the pipeline indeed results in an execution consistent with the tracking of MAETT. This approach is certainly viable (as they showed) but highly tedious. Subsequently, significant work has been done to automate invariant proving itself, e.g., by designing techniques called predicate abstraction (Lahiri et al. 2003; Saidi and Shankar 1999; Ray and Sumners 2007).

Reasoning About Memory Hierarchy

The term "memory hierarchy" refers to the hierarchy of cache levels, main memory, and virtual memory in a modern microprocessor system. The ISA models as described in Section "ISA Analysis" typically do not account for these features. Instead, a `LOAD` or `STORE` instruction is typically formalized as a direct and atomic access or update of the relevant memory location. This simplistic formalization is a high-level view of the ISA that is consistent with the programmer's expectation of the microprocessor functionality. On the other hand, this means that there is proof obligation to ensure that the memory systems developed in modern microprocessors indeed implement the abstraction provided by the ISA.

Consider a multiprocessor system shown where each processor has access to private L1 and L2 caches. An obvious problem for such systems is *cache coherence*, i.e., making sure that a read from a private cache would result in the same data being read as if the processor had read it (atomically) from the main memory. Concretely, consider the following simple memory system, which we call memory. The transition function of memory, memory.*next* is shown in pseudocode in Fig. 4.

Verification of a multiprocessor system with cache entails showing a correspondence between the executions of the microarchitectural model with the simple memory system discussed above. Architecturally, cache coherence is ensured by a protocol that allows a memory block to be loaded in the cache either (1) in read-only mode, in which case no private cache is allowed to have a writable copy of the block, or (2) in read-write mode, in which case only one private cache has (exclusive) access to the block and no other private cache would have a copy of the block. Unfortunately, these protocols tend to be highly complex. Indeed, verification of cache coherence protocols is an extremely active areas of research with several interesting approaches.

Theorem proving has played an active role in the verification of cache coherence. A key advantage of theorem proving in the application of such protocols has been to exploit the full power of formal mathematics and logic, to deal with the infinite states. The lemmas and proofs so obtained exactly show the invariants of the verified

Fig. 4 Pseudocode for state transition of system memory

```
a := i.addr
c := cline(a)
if (i.op == "store")
   mem[c].a.data := i.data
```

protocols and why the invariants hold. In contrast, algorithmic methods typically account for a small (finite) instance of the protocol, e.g., a system with (say) 4 processes. Nevertheless, the challenge with theorem applying theorem proving on memory protocols has been to come up with invariants. There have been two key directions to address this problem. In one direction, there have been ways to improve the automation of invariant discovery itself. This has resulted in a rich area of predicate abstraction (Graf and Saidi 1997; Lahiri et al. 2003; Ray and Sumners 2007). The key idea is to start with a set of predicates and apply the state transition of the system to identify how the predicate changes. For example, if one predicate specifies the property that a specific cache line is invalid, then a read request would cause the predicate to go from false to true. One can correspondingly develop an abstract state graph of the system which can be finite even if the target system one started with was unbounded. Another direction of research has been to formalize the protocols themselves in a different way to enable easy capture of invariants. For instance, instead of formalizing the state transition, the system can be formalized in terms of *flows* (Talupur et al. 2015).

Verification of Execution Units

The design of execution units, which include ALUs that implement integer, cryptographic, and floating-point operations, is datapath intensive. The verification of execution units entails proving the correctness of micro-operations or *uops* implemented in these units. The cost of undetected bugs in execution units can be enormous–Intel's FDIV "flaw" (Pratt 1995) is just one (in)famous example. It is difficult for random or even targeted simulations to achieve complete verification coverage for such designs. Additionally, the state space for such operations has increased over the years–for instance, x86's AVX512 instructions can have two or more 512-bit wide operands.

Theorem proving has been extensively used for verification of a variety of floating-point algorithms as well as their RTL implementations. ACL2 has been used for verifying floating-point units of its AMD's K5™and K7™processors (Moore et al. 1998; Russinoff 1998). In addition to the verification itself, this resulted in one of the most extensive formal treatments of computer arithmetic in the logic of a theorem prover (Russinoff 2018).

Many execution unit operations are verified using symbolic simulation–i.e., both the design and the specification are symbolically simulated on the same inputs, and then their resulting outputs are compared for equivalence. The good news is that modern off-the-shelf commercial model checkers and SAT solvers can achieve complete coverage of many fixed-cycle operations–indeed, verification of these operations can proceed automatically once the properties have been written (Pouarz and Agrawal 2016; Mukherjee et al. 2015, 2016). However, complex operations like multiplication (integer as well as floating-point), floating-point addition, fused multiply-addition, division, and square root still require human-assistance for a variety of reasons.

The verification of operations like multiplication, floating-point addition, and fused multiply-addition often requires *decomposition*. A common way to verify floating-point adders is to case-split primarily based on the exponent difference (i.e., near and far paths), and the signs of the operands (i.e., effective subtraction or addition)–special operands (e.g., NaN, infinity, etc.) are handled in a separate case (Chen and Bryant 1998; Russinoff 2000). The verification of integer and floating-point multiplication can be done by decomposing the design into several smaller pieces–e.g., the partial product generation and accumulation can be verified separately (Kaivola and Narasimhan 2001; O'Leary et al. 2013). There have been recent advances to automate the verification of multipliers by employing term rewriting inside a theorem prover (Temel and Hunt 2021) and by combining symbolic computer algebra and SAT solving (Kaufmann et al. 2019). Both these approaches involve identifying the "atomic blocks" in the design of a multiplier, namely, half and full adders, and then substituting them by their respective specifications–namely, the equations of the resulting sum and carry vectors expressed in terms of the inputs. This exposes the underlying regularity in the structure of the design, which is exploited for verification by either term substitution and/or bit-blasting using a SAT solver. Similarly, the verification of fused multiply-add can proceed by decomposing the design into two main phases: multiplication and addition followed by normalization and rounding; each of these two phases can then be decomposed further in the manner described above.

The verification of variable-cycle operations like divide and square root is more involved. Similar to floating-point addition, these operations also require case-splitting based on the kinds of input values– the latency varies by the case encountered (e.g., a divide-by-zero or a NaN input would lead to an early termination, normal and denormal numbers take different number of cycles, etc.). Additionally, since the common algorithms (e.g., restoring/non-restoring, SRT division, etc.) used to implement these operations are based on a recurrence relation, their verification involves discovering and proving inductive invariants. Correlation of the intermediate values in the design, like partial remainders and products, to the corresponding values in the specifications can also be tricky–this process requires considerable human insight and often significant input from the designer.

Given the operation- and implementation-specific nature of the human assistance required to verify these operations, interactive theorem provers like Forte, ACL2, and HOL light have been used successfully to prove their correctness. These verification efforts have led to the development of sophisticated libraries that enable proof reuse–these libraries mechanize a theory that defines fundamental arithmetic concepts and include proofs of key lemmas involved in reasoning about these operations (Russinoff 2018; Harrison 1999). Increasingly, the verification of these operations is being done by adopting a hybrid approach, where a theorem prover is used with an automated tool as a back-end solver–a user could use her insight to guide the theorem prover and come up with a broad proof "outline," which includes symbolically simulating the RTL and the specification functions, determining a good decomposition strategy if needed and discovering appropriate invariants; the automated tool can be used to discharge low-level and tedious proof

obligations (Swords and Davis 2011; Davis et al. 2014; Kaivola and Kohatsu 2003). This approach leverages the benefits of both interactive and automated verification techniques. Moreover, in case of decompositions, the theorem prover can be used to compose together all the individual cases–proved either by symbolic simulation inside the prover or with an automated tool–to make sure that there are no holes in coverage.

Deep Dive: Formalization and Analysis of (Simplified) x86

We now describe an ACL2 library called `x86isa` (Goel 2016) that models a subset of the x86 Instruction Set Architecture. This `x86isa` model is intended to provide a mathematical description of the x86 architecture at the same level of discourse as is found in the official Intel® Software Developer's manuals (Intel SDMs) (Intel Corporation 2020). Thus, it is a specification of the x86 ISA in the sense that it formalizes the *expected observable behavior* of an x86 processor. One can view `x86isa` as a simulator that can perform concrete as well as symbolic execution of x86 instructions, the latter of which is a crucial step toward general-purpose property verification. As such, this model can be used in a variety of projects in the fields of software, hardware, and compiler verification.

In this section, we discuss the design of `x86isa` and focus on some aspects that enable its role as a formal specification for x86 microarchitecture. We also discuss how this model can be used to test and verify properties of microarchitecture components by describing a recent application of `x86isa` toward verifying Centaur Technology's x86 instruction implementations (Goel et al. 2020, 2021; Goel and Sumners 2019). The goal here is to use `x86isa` as a case study to give the reader some insight into the development of models that are used for formal verification and to provide an overview of their general capabilities and limitations.

Approach

The `x86isa` model specifies the x86 ISA in a manner described earlier in Section "ISA Analysis"–the behavior of the machine is captured by an interpreter (i.e., an *ISA.run* function) that operates over the machine state. This interpreter essentially models the fetch-decode-execute cycle of an x86 processor.

The four core components of `x86isa` are the following:

Machine State: The machine state is modeled using a stobj whose fields describe the basic execution environment of an x86 processor (ref. Figure 3-2: 64-Bit Mode Execution Environment, Chapter 3, Intel SDMs). As of this writing, it contains the program counter, general-purpose registers, flags registers, floating-point registers, XMM/YMM/ZMM registers, segment registers, system table registers, control and debug registers, model-specific registers, and a memory model. Additionally, the machine state contains some fields that are an artifact of the model and not of the

x86 architecture–the purpose of these fields is to store information about the model's operation. For example, if any unimplemented x86 operation is encountered, then a special field called the *model status* is populated with appropriate information.

Instruction Specification Functions: An instruction specification function includes three main kinds of operations: reads from the machine state to fetch the operands, computation of the results using these operands, and writes of these results to the machine state. The core computational behavior is described by an *instruction semantic function*. For example, the pseudocode for the instruction semantic function of the ADD instruction is as shown in Fig. 5.

Step Function: The step function models a single step in the processor's execution, i.e., one fetch-decode-execute cycle. It takes an initial machine state as input and returns an updated state. The program counter contains the memory address of the instruction to be executed next; this instruction is fetched from the memory and if the fetch is successful, it is decoded. Then, control is dispatched to the appropriate instruction semantic function, which updates the machine state with the final effects of that instruction. If, at any point, an error is encountered, execution halts, and the also model status field is populated with an informative error message.

Run Function: The run function is the x86 ISA interpreter. It takes `n`, the number of steps to be executed and the initial machine state as inputs, and executes the step function `n` times or until an error or a halt instruction is encountered, whichever comes first. It returns the updated machine state.

The first step needed to simulate an x86 machine program on `x86isa` is to initialize the model state appropriately–i.e., store that program in the model's memory, populate the program counter, registers, and other memory locations, if necessary. Then, the run function is called. The final effects of the program can be observed by comparing the resulting machine state with the starting state. The contents of the machine state can be observed at any desired level of granularity–either after the execution of each instruction or a set of instructions, or at certain breakpoints, or when the run function terminates.

The values used to initialize the model state can either be concrete or symbolic. Concrete values are used when the model is used as an ISA simulator, and symbolic values are used when the model is used to perform formal analysis. Symbolic values allow the consideration of many, if not all, possible executions at the same time. Even parts of the program can be symbolic!

Fig. 5 Pseudocode of the ADD instruction

```
ADD(size, op1, op2, rflags):
// size = 8, 16, 32, 64
temp := op1[size-1:0] + op2[size-1:0]
ans := temp[size-1:0]
flags := add-rflags(size, op1, op2, rflags)
return(ans, flags)
```

Design Considerations

A useful specification should describe the behavior of the system it models in a way that is easy to understand; this description should be an accurate representation of the system. Additionally, it is helpful if specifications are accompanied by tools that facilitate their use as verification frameworks and simulators. The design and development of a specification can be done deliberately with these goals in mind.

The primary purpose of `x86isa` is to provide a mathematical specification of the x86 ISA that is suitable for use in formal analyses. We discuss how the `x86isa` library incorporates these goals and the role ACL2 plays toward achieving them.

Easy to Understand: In this project, the x86 ISA specification is simply executable code written in ACL2's programming language. Merely the executable nature goes a long way toward achieving this goal: one can simply run `x86isa` to observe what it does. Additionally, the code is documented, with both user- and developer-oriented topics accessible online (`x86isa: Documentation2022`). The sources which were consulted to write the specification functions are often cited in the source code–these citations point to relevant lines from specific sections of the Intel manuals. For many topics, the documentation is generated automatically from the code, which keeps them synchronized with each other. In further support of this goal, the specification functions are not optimized for execution efficiency (see Goal "Unified Model for Simulation and Formal Verification:" for a discussion regarding execution efficiency). Such optimizations can often complicate the algorithm implemented by that code, which can not only make the code difficult to comprehend but can also introduce bugs. As such, the specification functions implement a simple, straightforward algorithm. The simplicity of the specification functions is also what makes them amenable to be used in formal verification efforts–that is, these functions offer reasoning efficiency. Often, if a function is easy to understand, then it is easy to reason about.

Accurate: This goal is a little more difficult to address. How do we know that `x86isa` models the x86 ISA accurately? After all, code can have bugs, due to programming oversights or misunderstandings of the system being modeled. Again, the simplicity of the specification functions helps–simpler code is easier to review. However, the x86 ISA is a large and complex architecture, and code reviews can only take us so far. Ideally, though the formal model is *trusted*, we would like it to be *tested* as well. Another way to build confidence in the model's accuracy is by performing co-simulations–we compare the results of program runs on `x86isa` with those on a real x86 processor. (In an industrial setting, co-simulations can also be done against an internal "Golden Model".) This immediately raises an important point: if a formal model is validated using simulation, then any mathematical guarantees provided by using the formal model eventually depend on the guarantees offered by simulation. Why then is it worth using a formal model and incurring the overhead of theorem proving in the first place? The answer lies in separation of concerns. Formal models are designed for the express purpose of being specifications of the system under verification–their development is free

from the pressures to optimize code for execution, and the focus is entirely on capturing the system's behavior. Thus, the reliance on simulation for the validation of formal models is not as total as it may seem at first glance. Additionally, one can use ACL2 to prove properties about the formal model itself. For instance, one can prove noninterference theorems about specification functions that are not supposed to conflict with each other. As such, running as many simulations as possible is a beneficial exercise, and optimizing the specification functions for execution efficiency can make this a practical one. However, as we discussed earlier, supporting execution efficiency often comes at the expense of simplicity and reasoning efficiency. The following solution to this issue will be familiar by now: we can use ACL2 to prove that specification functions optimized for reasoning efficiency implement the same behavior as those optimized for execution efficiency, and thus, they can be used interchangeably. We simply use whichever definition is suitable, depending on the context. As discussed in Section "ACL2 Preliminaries", ACL2 provides many features that help in balancing this trade-off between reasoning and execution efficiency in a practical manner; `x86isa` heavily relies on features like `MBE`, guards, and *stobjs*.(We actually use an *abstract stobj* in `x86isa` to model the machine state. Abstract *stobjs* are an advanced ACL2 feature that use regular *stobjs* for concrete execution but allow an alternative definition (proven to correspond to the underlying stobj whose logical definition involves lists) to be used for reasoning.) To sum up, model validation can be effort-intensive–it can not only involve simulation but also theorem proving, both of which place inherently opposing demands on the model's design. However, validation is indispensable because it serves as evidence for the model's accuracy, thereby increasing confidence in the results of formal verification. This encourages adoption of the formal model by people other than its developers.

Unified Model for Simulation and Formal Verification: The `x86isa` model is accompanied by some tools that support its use as a simulator; these tools are especially useful for model validation via concrete simulations. Dynamic program instrumentation utilities allow monitoring the behavior of a running program in a manner similar to the GNU Debugger and Intel's Pin tool (Patil et al. 2004; Intel). For instance, one can step through a program one instruction at a time, conditionally or unconditionally trace reads from and writes to the machine state, insert breakpoints, log (requested parts of) the machine state to a file, and so on. One can even modify the program during its execution. We also have a binary program (ELF and Mach-O formats) loader library (`EXLD` 2022) in ACL2 that can parse an executable program file and load requested sections of it into the relevant addresses of the `x86isa`'s memory. This speeds up machine state initialization, which is the first step toward simulating a program on the model. This library also makes symbol table information available directly in ACL2, which, among other things, allows a user to instrument a machine code program by referring to subroutine names used in the source code instead of memory addresses that can change the next time the binary file is generated. The `x86isa` model is also accompanied by lemma libraries that support symbolic simulation. These lemmas are organized in collections to support various proof strategies that target different kinds of verification problems.

For instance, there are separate libraries that describe the interaction of reads and writes at the level of both physical and virtual memory. Note that all of these utilities for simulation and verification are written in ACL2 itself. An important advantage of writing such utilities in the same language as the formal specification is that one avoids any potential language-related cognitive dissonance issues that could cause unpredictable behavior that is difficult to debug.

Extensible Design: The x86 ISA is extended quite regularly. For instance, Intel maintains a document (Intel Corporation 2021) that describes features slated to be present in future x86 processors. The formal model must be able to keep up with this ever-expanding ISA design; adding a new feature must not incur an unreasonable manual overhead. In order to facilitate that, formal models must embrace the *extensible design principle* of software engineering. One way the `x86isa` model follows this principle is in its description of x86 instructions. The model contains a list, `inst.lst`, of all the instructions supported (or soon-to-be supported) by x86 processors. Each element of `inst.lst` is essentially a structure of product type that describes an instruction's encoding and lists the name of the ACL2 function that captures its semantics. Some fields of this structure are as follows:

1. **Mnemonic** can be a string (e.g., "`ADD`") or a keyword (e.g., `:2-BYTE-ESC` for byte 0x0F to indicate that it is the first byte of a two-byte opcode).
2. **Opcode** is a product type that describes an instruction variant of the mnemonic. It contains the prefixes (including `rex`, `vex`, and `evex`), opcode bytes, mode of operation, `ModR/M` byte (used for operand addressing and sometimes as an opcode extension), and CPUID feature flags.
3. **Operands** is a product type that describes the encoding of each operand of the instruction (e.g., the first operand is an `XMM` register indexed by the `reg` field of the `ModR/M` byte).
4. **Semantic Function** lists the name of the ACL2 function that describes the behavior of this instruction variant.
5. **Exceptions** list the conditions for detecting certain decode-time exceptions (e.g., throw an undefined operation exception if the `lock` prefix is present but unsupported by an instruction).

We generate many model functions from `inst.lst` automatically–e.g., functions to perform instruction decoding, dispatch control to appropriate semantic functions, etc. As such, adding support for a new x86 instruction is essentially a two-step process:

1. Writing and testing the semantic function that captures that instruction's behavior
2. Adding an appropriate instruction encoding entry, including the name of the above function, in `inst.lst`

Scope

The `x86isa` library does not model the entirety of the x86 ISA. As of this writing, only IA-32e and compatibility modes of operation are modeled in `x86isa`. Also,

it describes a single x86 core, and its memory model is sequentially consistent; as such, `x86isa` cannot be used to analyze concurrent behaviors. It does not model caches, translation look-aside buffers, exceptions and interrupts, and features like power management, virtual machine (VMX), and software guard extensions (SGX).

Despite these limitations, `x86isa` can still be used to perform a variety of useful analyses. For instance, `x86isa` can be used to do software verification–i.e., to establish the correctness of x86 machine-code programs w.r.t. a specification; an interested reader can refer to examples in this work (Goel 2016). The `x86isa` library has also been used at Centaur Technology to establish that the logic designers' Verilog/SystemVerilog implementation for instruction decoding has the same behavior as that of the `x86isa`'s decoder; more details are in Section "Application: Verifying x86 Instruction Implementations".

It should be noted that such missing features in `x86isa` are not due to some theoretical restrictions, but due to a lack of time and other such practical considerations. We typically support an ISA feature only when we undertake a verification project that requires a model of that feature.

Application: Verifying x86 Instruction Implementations

In this section, we focus on a verification project at Centaur Technology which uses `x86isa` to verify x86 instruction implementations; details of this project have been previously published elsewhere (Goel et al. 2020, 2021; Goel and Sumners 2019). The goal of this verification is to determine whether an x86 instruction is correctly implemented by its corresponding micro-operations. The focus is on the behavior of a single instruction; that is, the initial state is set up so that any interference with or impediments to the instruction's execution (e.g., due to interrupts, management of microarchitectural dependencies, etc.) are ruled out, which means that the instruction is known to run to completion.

The RTL design under verification includes the following:

- **decode** block: This block decodes bytes in the instruction stream in order to identify the x86 instruction. It is also responsible for detecting illegally encoded instructions and identifying the appropriate exceptions to be raised. For example, if an instruction is more than 15 bytes in length (the maximum allowed by the x86 ISA), then the **decode** block prescribes the `#GP` (general-purpose, interrupt 13) exception.
- **xlate** and **ucode** blocks: These blocks are responsible for the translation of a legal x86 instruction, obtained from the decode block, into micro-operations or *uops*. The **xlate** block translates an instruction to at most 6 *prelude* uops. There can also be an optional trap to a microcode ROM in the **ucode** block. The ROM contains a compressed version of the uops–also called *ROM instructions*–and these pass through a sub-block called the **microsequencer** that translates a ROM instruction into corresponding uops. All the uops, from **xlate** and **ucode**, that correspond to a legal x86 instruction are collectively referred to as the *ucode program*.

– **exec** block: The **exec** block has the RTL implementations of uops. This block may contain many different units–the ALU, for integer uops; the FPU, for floating-point uops, etc.

The "flow" of an instruction through these RTL blocks is assumed to proceed from a generalized, flushed legal state that allows that instruction to run to completion. The verification of uop scheduling and memory accesses is beyond the scope of this project.

An overview of this project is depicted in Fig. 6, where the three main theorems that state the correctness of the respective blocks are `decode-correct`, `xlate/ucode-correct`, and `exec-correct`.

The entirety of this project is done within ACL2 using many existing open-source tools and libraries. The RTL design is represented in ACL2 using VL (VL Verilog Toolkit; VL Verilog Toolkit: Documentation) and SV (SV A Hardware Verification Library; SV Documentation: A Hardware Verification Library), both of which are open-source ACL2 libraries. VL is used to translate SystemVerilog RTL designs into syntax trees, and SV takes these syntax trees, ascribes semantics to them, and produces next-state functions for the design signals. SV includes a tool called SVTV (SVTV) that can generate ACL2 functions corresponding to a multiphase symbolic simulation of the design. SVTVs facilitate initialization of the state, setting input signals, and sampling output and internal signals at appropriate phases. We refer to the RTL design blocks' representation in ACL2 by the names of the functions generated by VL/SV. Thus, the `sv-decode`, `sv-xlate`, `sv-ucode`,

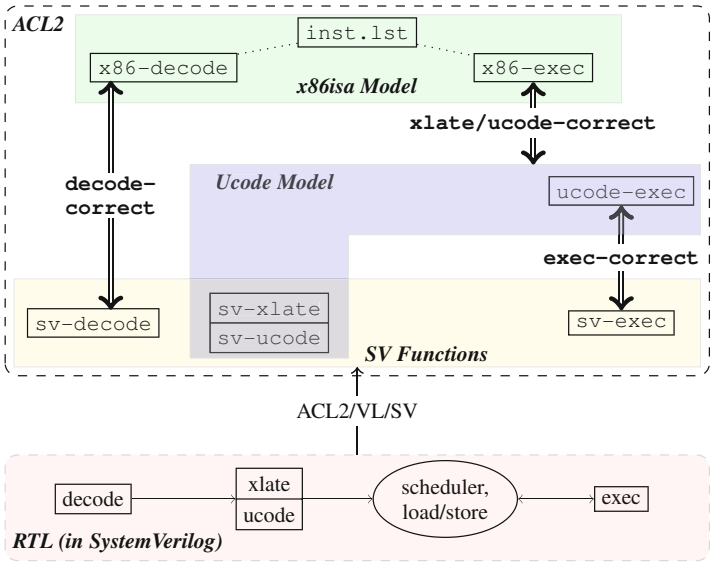


Fig. 6 Overview: Verifying Centaur's x86 instruction implementations

and `sv-exec` ACL2 functions correspond to the **decode**, **xlate**, **ucode**, and **exec** blocks, respectively. We note that VL and SV are trusted; i.e., we assume that these tools have been implemented correctly.

Open-source ACL2 library GL (Swords 2010; Swords and Davis 2011) is used for symbolic simulation. GL contains a prover that can verify ACL2 theorems involving finite domains (e.g., bit-vectors of a particular width). This prover is also verified in ACL2. GL symbolically simulates ACL2 function definitions to reduce finite ACL2 theorems into propositional logic formulas. These propositional logic formulas are then either proven using binary decision diagrams (BDDs) or simplified using And-Inverter Graph (AIG) algorithms and sent to a SAT solver. GL's successor FGL (Swords 2020) can also be used for symbolic simulation–FGL contains a sophisticated prover that can verify more general theorems than GL because it incorporates term rewriting alongside bit-blasting using SAT solvers.

Ucode Model

Similar to how the `x86isa` model formally specifies the x86 instruction set architecture, the ucode model formally specifies the microarchitecture. This model is also defined by an interpreter that acts on the ucode state–the effects of each uop are captured by the effects produced on the state by *uop semantic functions*. The ucode state can be thought of as extending the `x86isa` state; it not only captures the ISA state components but also the internal microarchitecture-specific registers, flags, and memory banks.

The definition of the program counter in the ucode model deserves special mention. Recall that the program counter of `x86isa` is simply the `rip` register; it contains the address of the x86 instruction that the model is poised to execute. Analogously, the program counter in the ucode model is a data structure that consists of the prelude uops and, if applicable, an address in the microcode ROM–thus, the program counter essentially contains a ucode program that corresponds to the x86 instruction that the ucode model is poised to execute. A uop is represented by a product-type consisting of all the information needed to identify the uop–the uop's opcode, source and destination locations, immediate data, and so on. The value of this program counter can be obtained by simulating the `sv-xlate` design function; given a description of the x86 instruction, this function generate uops and, if needed, a ROM trap address. More details are in Section "Verification of the **Xlate/Ucode** Blocks".

The ucode model begins execution by reading the first element in the program counter. If it is a prelude uop, then its corresponding uop semantic function is executed, after which that uop is removed from the program counter. Thus, unless there are traps to the microcode ROM, the ucode model halts execution when there are no uops left in the program counter. When a trap is present, the microcode ROM is read to obtain the ROM instruction at that address, and then, the **microsequencer** block is simulated, via the `sv-ucode` design functions, to get the corresponding uops, which are then executed by calling their respective uop semantic functions, as with prelude uops. A terminal ROM instruction (i.e., the last ROM instruction in a

ucode program) is tagged with a `.T` label, which signals to the ucode model that the x86 instruction has run to completion and it then halts execution.

The uop semantic functions used in the ucode model are used to verify the uop's RTL implementations in the **exec** block–i.e., `exec-correct` theorem; this is discussed in Section "Verification of the **exec** Block". These functions are also used in the proof of the `xlate/ucode-correct`–see Section "Verification of the **Xlate/Ucode** Blocks".

Verification of the exec Block

The **exec** block contains many execution sub-modules, each of which performs different kinds of operations (e.g., floating-point and integer operations are usually implemented in separate modules). The uops are verified, using techniques previously presented in section "Verification of Execution Units", in the scope of the **exec** block–the corresponding ACL2 design function is `sv-exec`. Working with the **exec** block instead of the sub-modules affords the convenience of a stable block interface, which in turn makes it possible to write uop semantic functions in a uniform manner that is reusable across many chip projects.

A Candidate Instruction

The first step involved in verifying an x86 instruction implementation is to pick the instruction. This may sound obvious, but it is not a straightforward proposition. For instance, using the mnemonic to identify a candidate instruction can be problematic–the x86 ISA often has multiple variants for an instruction with the same mnemonic. For example, the double-shift instruction `SHRD` corresponds to the variants listed in Table 1; note that there are two separate opcodes, each of which describes six distinct variants. Moreover, many instruction byte sequences can correspond to a single variant–for instance, `rex` and operand-size override prefixes are ignored in an instruction executing a byte operation; the instruction variants with and without these prefixes are operationally the same.(We assume that these variants are of legal length; specifically, adding these prefixes did not increase the length of the x86 instruction beyond the legal limit.) Additionally, there are multiple configuration settings, like the operating mode of the processor, to take into account for each variant.

The RTL implementation is a big factor in deciding how these variants are picked for verification. For the **decode** block, the proof of correctness of all the variants

Table 1 `SHRD` variants

Opcode 0x0F_AC	Opcode 0x0F_AD
`SHRD r16, r16, imm8`	`SHRD r16, r16, CL`
`SHRD m16, r16, imm8`	`SHRD m16, r16, CL`
`SHRD r32, r32, imm8`	`SHRD r32, r32, CL`
`SHRD m32, r32, imm8`	`SHRD m32, r32, CL`
`SHRD r64, r64, imm8`	`SHRD r64, r64, CL`
`SHRD m64, r64, imm8`	`SHRD m64, r64, CL`

which have the same opcode is usually clubbed together. One may do additional case-splits based on some RTL-specific internal parameters. For the **xlate/ucode** blocks, a variant becomes a "stand-alone" verification target if it has exactly one ucode program corresponding to it. For example, if `SHRD r16, r16, imm8` and `SHRD r32, r32, imm8` are implemented by the same sequence of uops, then they are considered to be the same variant for the purposes of verifying these blocks. An advantage of this choice over, say, clubbing all the possible instruction variants together is that the microcode program under verification will have only data-driven control paths (e.g., an early exit if `imm8` is zero) instead of control paths dictated by a specific variant (e.g., jump to another block of uops for variant `A` and yet another block for variant `B`, and so on.). This means that there will be fewer case-splits during a proof, which not only speeds up verification but also makes it more amenable to automation using techniques like bit-blasting.

We will illustrate our verification approach for the **decode**, **xlate**, and **ucode** blocks by using a running example of the following `SHRD` variant. This instruction's destination and source are 64-bit registers whose indices are represented by `<dreg64>` and `<sreg64>`; it also takes an immediate byte `<imm8>` to specify the shift amount.

```
variant: SHRD <dreg64>, <sreg64>, <imm8>
bytes:   0x48 0x0F 0xAC 0b11<dreg64><sreg64> <imm8>
```

The byte `0x48` is the `rex` prefix, which indicates that this is a 64-bit operation. The bytes `0x0F 0xAC` represent the opcode. The byte `0b11<dreg64><sreg64>` represents the `ModR/M`, whose `mod` field is `0b11`, which indicates that the `r/m` field denotes a register operand. The `r/m` field is a 3-bit value `<dreg64>`, and the `reg` field is a 3-bit value `<sreg64>`.

Our ACL2 specification function `shrd-spec` (from the `x86isa` model) describes the behavior of this instruction. The semantics of this instruction variant are as follows: the destination is shifted right by a value indicated by the given immediate byte (masked by `0x3F`) and the resulting empty bit positions are filled with bits shifted in from the source (least-significant bit first). Though `SHRD` can also affect the `rflags` register, we omit all flag-related discussions here.

Figure 7 shows an example of a concrete run of this instruction variant, and Tables 2 and 3 show the corresponding uops that implement this variant, along with a log of the computation performed by each uop for this particular concrete run.

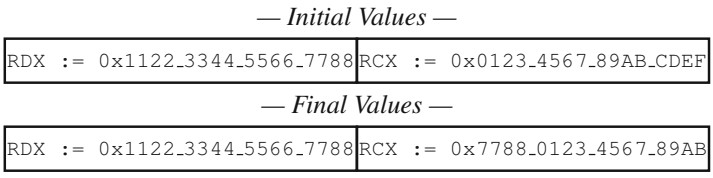


Fig. 7 An example of a concrete run of `SHRD RCX, RDX, 16`: the destination register `RCX` is shifted right by 16 and the low 16 bits of `RDX` are shifted in from the left. `RDX` remains unchanged

Table 2 SHRD prelude uops: Output from the **xlate** block

Uop	Concrete Run (ref. Fig. 7) & Description
`MOVSX G2, RCX` `(SSZ: 64; DSZ: 64)`	`G2 ← 0x0123_4567_89AB_CDEF` *Move RCX to internal register G2*
`MOVZX G3, <imm8>` `(SSZ: 8; DSZ: 64)`	`G3 ← 16` *Move immediate to internal by 0x3F*

Table 3 SHRD uops in the microcode ROM

`AND G3, G3, 63` `(SSZ: 8; DSZ: 64)`	`G3 ← 16` *Mask immediate operand*
`MOV G10, -1` `(SSZ: 64; DSZ: 64)`	`G10 ← 0xFFFF_FFFF_FFFF_FFFF` *Move -1 to internal register G10*
`JE G3, 0, ent_nop` `(SSZ: 16; DSZ: 16)`	`No jump taken` *Jump to routine ent_nop if G3 == 0*
`SUB G5, 0, G3` `(SSZ: 32; DSZ: 32)`	`G5 ← 0xFFFF_FFF0; ZF ← 0` *Store -G3 in internal register G5; clear the zero flag because result is non-zero*
$\text{SHR}_{<!ZF>}$ `G10, G10, G5` `(SSZ: 64; DSZ: 64)`	`G10 ← 0xFFFF` *Shift G10 right by (G5 & 63) if ZF == 0*
$\text{AND}_{<ZF>}$ `G10, G10, 0` `(SSZ: 64; DSZ: 64)`	`G10 ← 0xFFFF` *Set G10 to 0 if ZF == 1*
`AND G6, RDX, G10` `(SSZ: 64; DSZ: 64)`	`G6 ← 0x7788` *Store (RDX & G10) in internal register G6*
`SHR G7, G2, G3` `(SSZ: 64; DSZ: 64)`	`G7 ← 0x0000_0123_4567_89AB` *Store (G2 » G3) in G7*
`SHL G2, G7, G3` `(SSZ: 64; DSZ: 64)`	`G2 ← 0x0123_4567_89AB_0000` *Store (G7 « G3) in G2*
`OR G2, G2, G6` `(SSZ: 64; DSZ: 64)`	`G2 ← 0x0123_4567_89AB_7788` *Store (G2 | G6) in G2*
`ROR G7, G2, G3` `(SSZ: 64; DSZ: 64)`	`G7 ← 0x7788_0123_4567_89AB` *Rotate G2 right by G3 and store result in G7*
`OR RCX, G7, G7` `(SSZ: 64; DSZ: 64)`	`RCX ← 0x7788_0123_4567_89AB` *Store the result of G7 | G7 in RCX*

Verification of the Decode Block

The verification of the **decode** block entails proving that this block correctly detects whether an incoming byte sequence corresponds to a valid or to an illegal x86 instruction. We call this property `decode-correct`; see Fig. 6. The `x86-decode` specification function defines the expected behavior of this block, and the RTL implementation under verification is represented by the design function `sv-decode`. Note that for this proof, the "variable" components of the instruction byte sequence (e.g., register indices) are symbolic and the fixed parts (e.g., opcode) are concrete. This ensures all possible invocations of the variant are covered.

For a byte sequence representing a legal instruction, the function `x86-decode` identifies all the instruction components and populates a valid-instruction data

structure with these components accordingly. The `inst.lst` structure is consulted during this parsing to determine if certain bits/bytes (e.g., the `ModR/M` or immediate data) are expected for that particular opcode. Note that `x86-decode` does not check exceptions that occur in the **xlate/ucode** or **exec** blocks (e.g., arithmetic underflow/overflow exceptions) or in a block which is currently beyond the scope of this project (e.g., page faults).

The only nontrivial decode-time exception for our `SHRD` example occurs when a `LOCK` prefix is present, in which case it is checked whether the **decode** block prescribes a `#UD` (undefined operation) exception, as dictated by `x86-decode`. For a legal `SHRD` instruction, case-splitting on the opcode value and a few internal parameters of the **decode** block results in a few thousand cases. These are proved via GL/SAT solving in around 5–10 seconds each, and the entire instruction's proof can be done across multiple machines in parallel.

Verification of the Xlate/Ucode Blocks

In order to verify that the RTL implements legal x86 instructions correctly, we need to prove that the ucode program produced by the **xlate/ucode** blocks has the same observable behavior as that of the x86 instruction. We call this property `xlate/ucode-correct`, and the RTL implementation is represented by the design functions `sv-xlate` and `sv-ucode`. The implementation of the **xlate/ucode** blocks changes often; as such, instead of defining their specification functions, one can simply symbolically simulate their design functions for the instruction variant under verification and prove that the resulting uops correctly implement that instruction. Note that the proof of `xlate/ucode-correct` involves not only the **xlate/ucode** blocks, which produce the uops, but also the **exec** block, which implement these uops. However, instead of using the `sv-exec` design function, the uop semantic functions `ucode-exec` are used here–these are the same functions that are used in **exec** block verification–see Section "Verification of the **exec** Block". This is justified because for each uop, `sv-exec` is proven to correspond to these functions; this decision makes the proof of `xlate/ucode-correct` tractable because `ucode-exec` functions are much simpler than their corresponding RTL implementations.

The design functions `sv-xlate` and `sv-ucode` expect a populated valid-instruction data structure as input and produce the corresponding uops as the output of symbolic simulation. One can obtain this instruction data structure by symbolically simulating `sv-decode`, given an appropriately symbolic byte sequence, as discussed previously in Section "Verification of the **Decode** Block". However, the results (i.e., appropriately populated valid-instruction data structure) from the verification of the **decode** block cannot always be reused for this proof because those might correspond to a different instruction variant from the one under verification here; the case-splits done for the **decode** block verification can differ from those done for these blocks–see Section "A Candidate Instruction" for details. One could directly supply a symbolic byte sequence to `sv-decode`, but that can be tedious, especially for instructions that have complicated encoding. Instead, with the help of the `inst.lst` structure, one can pick and encode an x86 instruction

that can then be input to `sv-decode` to obtain the valid-instruction data structure. For instance, the following indicates the relevant variant of the `SHRD` instruction of our running example:

```
Mnemonic:   SHRD   Opcode:   0x0F_AC
Variant:    Size := 64; OP1 := GPR
Mode:       64-bit mode
Symbolic:   OP1, OP2, OP3
```

The first line is used to find the appropriate entry for `SHRD` in `inst.lst`, which gives information about the arity and kinds of operands of this instruction. The second line helps in picking the variant under verification by specifying the operation width, 64, and by constraining the first operand to be a register (i.e., not a memory location). The third line picks the machine configuration, and the fourth line instructs the framework to pick a symbolic value for the operands–the register indices and the immediate byte. All of this information is used to populate the instruction data structure.

This structure is then passed through `sv-xlate` and `sv-ucode`, and thus, one obtains the ucode program which comprises the prelude uops (i.e., uops generated by **xlate**) and the address of the ucode routine, if applicable. One can then attempt to prove that all relevant executions of this ucode program implement the `SHRD` instruction. That is, the effects produced by the instruction specification function `shrd-spec` on the ISA-visible components of the ucode state are the same as those produced by the implementation (i.e., uops' execution), provided that the arguments of `shrd-spec` correspond to the instruction's operands. These kinds of proofs can be done by techniques like the the clock function approach, stepwise invariant method (Boyer and Moore 1996; Ray and Moore 2004; Ray et al. 2008), and decompilation-into-logic (Myreen et al. 2008; ACL2 Books: Codewalker 2014), all of which could employ either GL /SAT's automatic bit-blasting or ACL2 rewriting. The central idea is the symbolic simulation of these uops on our ucode model.

Note that being able to obtain the prelude uops and the trap address automatically allows one to keep up with the constantly changing RTL. For nonmajor "everyday" changes in the RTL (e.g., if the ROM addresses change or if the uops use different internal registers), the proofs usually work without any intervention.

Discussion

The goals of this project are to verify the decode, translate, microsequencer, and execution blocks. This project does not take load/store units, caches, register mapping, scheduler, etc., into account, and, as such, offers no guarantees there. A benefit of this approach is that it enables a divide-and-conquer strategy for verifying instruction implementations. The verification of the **decode** block, the **xlate/ucode** blocks, and the **exec** blocks can all be done independently of each other. Indeed, the verification of the **exec** block has been done at Centaur regularly for over a decade, and this project incorporated and extended all that work. Another benefit is that one does not need to formally specify (or even understand) the **xlate** and **ucode** blocks–one can simply symbolically simulate these units and reason about their outputs

(i.e., uops). This lends robustness to the process–design changes to those blocks (apart from interface changes) do not impede formal verification.

Theorem Proving Beyond Microarchitecture

Our focus in this chapter has been on architecture and microarchitecture assurance. Of course, the scope of theorem proving goes far beyond that. A treatment of the role of theorem proving in assurance of computing systems would be incomplete without at least a passing mention of some of these topics.

- **Software Verification:** Theorem proving has been used extensively in software verification. This has ranged from high-level programs to low-level binary implementations. One of the most impressive recent verification efforts has been the seL4 microkernel, where a complete operating system microkernel has been formally proven correct (Winwood et al. 2010). Theorem proving has also been used to verify compilers (Moore 1996; Leroy 2006).
- **Analog Systems:** Theorem proving has been successfully used for verification of various analog and mixed-signal components. ACL2 in particular has been used for verification of flash memories (Ray and Bhadra 2007; Ray et al. 2010).
- **Concurrent Protocols:**. Theorem proving has been successfully used for verification of synchronization protocols, memory protocols, non-blocking concurrent data structure implementations, checkpointing algorithms, and others (Ray and Sumners 2013; Moore and Porter 2002; Russinoff 1994).

Conclusion

We have provided a broad overview of theorem proving and its applications to verification of computing systems. We discussed how one approaches such verification with theorem proving as well as the complexities in the process of proving nontrivial properties of complex computing systems. The complexities include the notions of correctness to be used, difficulties in formalization, and managing the complexity of proofs themselves. We illustrated the use of theorem proving in the verification of a nontrivial machine architecture, e.g., a detailed model of x86.

Of course, theorem proving has been extensively used in formalization and verification of computing systems for the last five decades with a rich body of impressive achievements over the years. It is impossible to do justice to this topic within the constraints of a single chapter. This chapter should be taken as a scratch in the surface of this topic. We hope it will provide a broad–if high-level–picture of the area and provide an entry point to beginning researchers into its large and extensive literature.

References

Aagaard M, Cook B, Day N, Jones RB (2001) A framework for microprocessor correctness statements. In: Margaria T, Melham TF (eds) Proceedings of the 11th International Conference on Correct Hardware Design and Verification Methods (CHARME 2001). LNCS, vol 2144. Springer, Scotland, pp 443–448

Aagard MD, Jones RB, Kaivola R, Kohatsu KR, Seger CH (2000) Formal verification of iterative algorithms in microprocessors. In: Proceedings of the 37th ACM/IEEE Design Automation Conference (DAC 2000). ACM Press, Los Angeles, pp 201–206

ACL2 Books: Codewalker. Online; accessed: Feb 2022. Github, (2014) https://github.com/acl2/acl2/tree/master/books/projects/codewalker

Arm ISA Specifications. Online. https://developer.arm.com/architectures/cpu-architecture/a-profile/exploration-tools

Armstrong A, Bauereiss T, Campbell B, Reid A, Gray KE, Norton RM, Mundkur P, Wassell M, French J, Pulte C, Flur S, Stark I, Krishnaswami N, Sewell P (2019) Isa semantics for armv8-a, risc-v, and cheri-mips. Proc ACM Program Lang 3. pp 1–31, https://doi.org/10.1145/3290384

Bauereiss T, Campbell B, Sewell T, Armstrong A, Esswood L, Stark I, Barnes G, Watson RNM, Sewell P (2021) Verified security for the morello capability-enhanced prototype arm architecture. Technical Report UCAM-CL-TR-959, University of Cambridge, Computer Laboratory

Bevier WR, Hunt WA Jr, Moore JS, Young WD (1989) Special issue on system verification. J Autom Reason 5(4):409–530

Boyer RS, Kaufmann M, Moore JS (1995) The Boyer-Moore theorem prover and its interactive enhancements. Comput Math Appl 29(2):27–62

Boyer RS, Moore JS (1996) Mechanized formal reasoning about programs and computing machines. Automated reasoning and its applications: essays in honor of larry wos, pp 147–176 . https://www.cs.utexas.edu/users/boyer/bm96.pdf

Boyer RS, Moore JS (2002) Single-threaded objects in ACL2. In: Krishnamurthy S, Ramakrishnan CR (eds) Practical Aspects of Declarative Languages (PADL). LNCS, vol 2257. Springer, pp 9–27

Boyer RS, Yu Y (1996) Automated proofs of object code for a widely used microprocessor. J ACM 43(1):166–192. http://dl.acm.org/citation.cfm?id=227603

Bronstein A, Talcott TL (1990) Formal verification of pipelines based on string-functional semantics. In: Claesen LJM (ed) Formal VLSI correctness verification. VLSI design methods II, pp 349–366

Burch JR, Dill DL (1994) Automatic verification of pipelined microprocessor control. In: Dill DL (ed) Proceedings of the 6th International Conference on Computer-Aided Verification (CAV 1994). LNCS, vol 818. Springer, pp 68–80

Chen YA, Bryant RE (1998) Verification of floating-point adders. In: International Conference on Computer Aided Verification. Springer, pp 488–499

Church A, Kleene SC (1937) Formal definitions in the theory of ordinal numbers. Fundam Math 28:11–21

CLHS (Common Lisp HyperSpec) Online; accessed: 2022 http://www.lispworks.com/reference/HyperSpec/index.html

Davis J, Slobodova A, Swords S (2014) Microcode verification–another piece of the microprocessor verification puzzle. In: International Conference on Interactive Theorem Proving. Springer, pp 1–16

Degenbaev U (2012) Formal specification of the x86 instruction set architecture. Ph.D. thesis, Universität des Saarlandes. http://rg-master.cs.uni-sb.de/publikationen/UD11.pdf

Dowek G, Felty A, Huet G, Paulin C, Werner B (1991) The coq proof assistant user guide version 5.6. Technical Report TR 134, INRIA

EXLD: ELF and Mach-O File Parser, Documentation. Online; accessed: 2022. https://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=EXLD____EXECLOADER

Floyd R (1967) Assigning meanings to programs. In: Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematcs, vol XIX. American Mathematical Society, Providence, pp 19–32

Fox A (2015) Improved tool support for machine-code decompilation in HOL4. In: International Conference on Interactive Theorem Proving. Springer, pp 187–202

Goel S (2016) Formal verification of application and system programs based on a validated x86 ISA model. Ph.D. thesis, Department of Computer Science, The University of Texas at Austin. https://repositories.lib.utexas.edu/handle/2152/46437

Goel S, Slobodova A, Sumners R, Swords S (2020) Verifying x86 instruction implementations. In: Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020. Association for Computing Machinery, New York, pp 47–60. https://doi.org/10.1145/3372885.3373811

Goel S, Slobodova A, Sumners R, Swords S (2021) Balancing automation and control for formal verification of microprocessors. In: Silva A, Leino KRM (eds) Computer Aided Verification. Springer International Publishing, Cham pp 26–45

Goel S, Sumners R (2019) Using x86isa for microcode verification. In: SpISA 2021: Workshop on Instruction Set Architecture Specification. https://www.cl.cam.ac.uk/~jrh13/spisa19/paper_08.pdf

Goldstein HH, von Neumann J (1961) Planning and coding problems for an electronic computing instrument. In: von Neumann J (ed) Collected Works, vol V. Pergamon Press, Oxford

Gordon MJC, Melham TF (eds) (1993) Introduction to HOL: a theorem-proving environment for higher-order logic. Cambridge University Press, ISBN 0-521-44189-7. Journal of Functional Programming, **4**(4), pp 557–559. https://doi.org/10.1017/S0956796800001180

Graf S, Saidi H (1997) Construction of abstract state graphs with PVS. In: Grumberg O (ed) Proceedings of the 9th International Conference on Computer-Aided Verification (CAV 1997). LNCS, vol 1254. Springer, pp 72–83

Greve D, Wilding M, Hardin D (2000) High-speed, analyzable simulators. In: Kaufmann M, Manolios P, Moore JS (eds) Computer-aided reasoning: ACL2 case studies, Kluwer Academic Publishers, Boston, pp 89–106

Greve DA (1998) Symbolic simulation of the JEM1 microprocessor. In: Gopalakrishnan G, Windley P (eds) Formal methods in computer-aided design. Lecture notes in computer science, vol 1522. Springer, Berlin/Heidelberg, pp 321–333. https://doi.org/10.1007/3-540-49519-3_21

Greve DA, Kaufmann M, Manolios P, Moore JS, Ray S, Ruize-Reina JL, Sumners R, Vroon D, Wilding M (2008) Efficient execution in an automated reasoning environment. J Funct Program 18(1):15–46

Harrison J (1999) A machine-checked theory of floating point arithmetic. In: International Conference on Theorem Proving in Higher Order Logics. Springer, pp 113–130

He J, Hoare CAR, Fränzle M, Müller-Olm M, Olderog ER, Schenke M, Hansen MR, Ravn AP, Rischel H (1994) Provably correct systems. In: International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems. Springer, pp 288–335

Hunt WA Jr (1989)Microprocessor design verification. J Autom Reason 5(4):429–460. http://www.cs.utexas.edu/~boyer/ftp/cli-reports/048.pdf

Hunt WA Jr (1994) FM8501: a verified microprocessor. LNAI, vol 795. Lecture Notes in Artificial Intelligence, Springer, ISBN: 9783540579601

Intel: Pin: A Dynamic Binary Instrumentation Tool. http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

Intel Corporation (2021) Intel® Architecture Instruction Set Extensions Programming Reference. Online. Order Number: 319433-044. https://software.intel.com/en-us/articles/intel-sdm

Intel Corporation (2020) Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4. Online. Order Number: 325462-072USs. https://software.intel.com/en-us/articles/intel-sdm

Kaivola R, Kohatsu K (2003) Proof engineering in the large: formal verification of Pentiumő 4 floating-point divider. Int J Softw Tools Technol Transfer 4(3):323–334

Kaivola R, Narasimhan N (2001) Formal verification of the Pentiumő 4 multiplier. In: Sixth IEEE International High-Level Design Validation and Test Workshop, pp 115–120. https://doiu.org/10.1109/HLDVT.2001.972817

Kaufmann D, Biere A, Kauers M (2019) Verifying large multipliers by combining sat and computer algebra. In: 2019 Formal Methods in Computer Aided Design (FMCAD). IEEE, pp 28–36

Kaufmann M, Manolios P, Moore JS (eds) (2000a) Computer-aided reasoning: ACL2 case studies. Kluwer Academic Publishers, Boston

Kaufmann M, Manolios P, Moore JS (2000b) Computer-aided reasoning: an approach. Kluwer Academic Publishers, Boston

Kaufmann M, Moore JS (1994) Design goals of ACL2. Technical Report 101, Computational Logic Incorporated (CLI), Austin

Kaufmann M, Moore JS (1997) A precise description of the acl2 logic. See https://www.cs.utexas.edu/users/moore/publications/km97a.pdf

Lahiri SK, Bryant RE, Cook B (2003) A symbolic approach to predicate abstraction. In: Hunt WA Jr, Somenzi F (eds) Proceedings of the 15th International Conference on Computer-Aided Verification. LNCS, vol 2275. Springer, pp 141–153

Leroy X (2006)Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: Proceedings of the 33rd Symposium on Principles of Programming Languages (POPL 2006). ACM Press, pp 42–54

Levy HM (1984) Capability-based computer systems. Butterworth-Heinemann, Newton

Liu H, Moore JS (2004) Java program verification via a JVM deep embedding in ACL2. In: International Conference on Theorem Proving in Higher Order Logics. Springer, pp 184–200

Manolios P (2000) Correctness of pipelined machines. In: Hunt WA Jr, Johnson SD (eds) Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD 2000), LNCS, vol 1954. Springer, Austin, pp 161–178

Manolios P, Vroon D (2003) Algorithms for ordinal arithmetic. In: Baader F (ed) Proceedings of the 19th International Conference on Automated Deduction (CADE 2003). LNAI, vol 2741. Springer, Miami, pp 243–257

Moore JS (1996) Piton: a mechanically verified assembly-level language. Automated reasoning series, Kluwer Academic Publishers, USA

Moore JS (2003) Proving theorems about Java and the JVM with ACL2. In: Broy M, Pizka M (eds) Models, algebras, and logic of engineering software. IOS Press, pp 227–290

Moore JS, Lynch T, Kaufmann M (1998) A mechanically checked proof of the kernel of the AMD5K86 floating-point division algorithm. IEEE Trans Comput 47(9):913–926

Moore JS, Porter G (2002) The apprentice challenge. ACM Trans Program Lang Syst (ACM TOPLAS) 24(3):1–24

Mukherjee R, Joshi S, Griesmayer A, Kroening D, Melham T (2016) Equivalence checking of a floating-point unit against a high-level c model. In: Fitzgerald J, Heitmeyer C, Gnesi S, Philippou A (eds) FM 2016: Formal Methods. Springer International Publishing, Cham, pp 551–558

Mukherjee R, Kroening D, Melham T, Srivas M (2015) Equivalence checking using trace partitioning. In: 2015 IEEE Computer Society Annual Symposium on VLSI, pp 13–18. https://doi.org/10.1109/ISVLSI.2015.110

Myreen MO, Gordon M, Slind K (2008) Machine-code verification for multiple architectures – An application of decompilation into logic. In: Formal methods in computer-aided design, 2008. FMCAD'08, pp 1–8. https://doi.org/10.1109/FMCAD.2008.ECP.24, http://www.cl.cam.ac.uk/~mom22/decomp.pdf

Nipkow T, Paulson LC, Wenzel M (2002) Isabelle/HOL: a proof assistant for higher-order logic, vol 2283. Springer Science & Business Media, Lecture Notes in Computer Science, Springer Berlin. https://doi.org/10.1007/3-540-45949-9

O'Leary J, Kaivola R, Melham T (2013) Relational ste and theorem proving for formal verification of industrial circuit designs. In: 2013 Formal Methods in Computer-Aided Design. IEEE, pp 97–104

Owre S, Rushby JM, Shankar N (1992) PVS: a prototype verification system. In: Kapur D (ed) 11th International Conference on Automated Deduction (CADE). Lecture notes in artificial intelligence, vol 607. Springer, Saratoga, pp 748–752

Patil H, Cohn R, Charney M, Kapoor R, Sun A, Karunanidhi A (2004) Pinpointing representative portions of large intel ő itanium ő programs with dynamic instrumentation. In: 37th International Symposium on Microarchitecture (MICRO-37'04), pp 81–92. https://doi.org/10.1109/MICRO.2004.28

Paulson L (1993) Set theory for verification: I. From foundations to functions. J Autom Reason 11:353–389

Paulson L (1995) Set theory for verification: II. Induction and recursion. J Autom Reason 15:167–215

Pouarz TW, Agrawal V (2016) Efficient and exhaustive floating point verification using sequential equivalence checking. DVCon

Pratt VR (1995) Anatomy of the pentium bug. In: Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development, TAPSOFT'95. Springer, Berlin/Heidelberg, pp 97–107

Ray S, Bhadra J (2007) A mechanized refinement framework for analysis of custom memories. In: Baumgartner J, Sheeran M (eds) Proceedings of the 7th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2007). IEEE Computer Society, Austin, pp 239–242

Ray S. Bhadra J, Portlock T, Syzdek R (2010)Modeling and verification of industrial flash memories. In: Inernational Symposium on Quality Electronic Designs

Ray S, Hunt WA Jr, Matthews J, Moore JS (2008) A mechanical analysis of program verification strategies. J Autom Reason 40(4):245–269

Ray S, Moore JS (2004) Proof styles in operational semantics. In: Hu AJ, Martin AK (eds) Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2004). LNCS, vol 3312. Springer, Austin, pp 67–81

Ray S, Sumners R (2007) Combining theorem proving with model checking through predicate abstraction. IEEE Des Test Comput 24(2):132–139

Ray S, Sumners R (2013) Specification and verification of concurrent programs through refinements. J Autom Reason 51(3):241–280

Reid A (2016) Trustworthy specifications of ARM v8-A and v8-M system level architecture. In: Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design (FMCAD'16)

Reid A, Chen R, Deligiannis A, Gilday D, Hoyes D, Keen W, Pathirane A, Shepherd O, Vrabel P, Zaidi A (2016) End-to-end verification of processors with ISA-formal. In: International Conference on Computer Aided Verification. Springer, pp 42–58

Russinoff D (1992) A mechanical proof of quadratic reciprocity. J Autom Reason 8:3–21

Russinoff D (1994) A mechanically verified incremental garbage collector. Form Asp Comput 6:359–390

Russinoff D (1998) A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions. LMS J Comput Math 1:148–200

Russinoff DM (2000) A case study in formal verification of register-transfer logic with acl2: The floating point adder of the amd athlon tm processor. In: International Conference on Formal Methods in Computer-Aided Design. Springer, pp 22–55

Russinoff DM (2018) Formal verification of floating-point hardware design: a mathematical approach. Springer, Springer International Publishing, ISBN: 9783319955131

Saidi H, Shankar N (1999) Abstract and model check while you prove. In: Halbwacha N, Peled D (eds) Proceedings of the 11th International Conference on Computer-Aided Verification (CAV 1999), LNCS, vol 1633. Springer, pp 443–453

Sawada J, Hunt WA Jr (2002a) Verification of FM 9801: An out-of-order microprocessor model with speculative execution, exceptions, and program-modifying capability. Formal Meth Syst Des 20(2):187–222

Sawada J, Hunt WA Jr (2002b) Verification of FM 9801: An Out-of-Order Microprocessor Model with Speculative Execution, Exceptions, and Program-Modifying Capability. Formal Meth Syst Des 20(2):187–222 http://dl.acm.org/citation.cfm?id=584665

Shankar N (1997) Metamathematics, machines, and gödel's proof. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press. ISBN: 9780521585330

Srivas M, Bickford M (1990) Formal verification of a pipelined microprocessor. IEEE Softw 7(5):52–64

SV Documentation: A Hardware Verification Library. Online (Accessed: 2022). http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____SV

SV: A Hardware Verification Library. Online (Accessed: 2022). https://github.com/acl2/acl2/tree/master/books/centaur/sv

SVTV: A Structure for Simulation Pattern of a Hardware Design. Online (Accessed: 2022). http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____DEFSVTV

Swords S (2010) A verified framework for symbolic execution in the ACL2 theorem prover. Ph.D. thesis, Department of Computer Science, The University of Texas at Austin. http://repositories.lib.utexas.edu/handle/2152/ETD-UT-2010-12-2210

Swords S (2020) New rewriter features in fgl. Electronic Proceedings in Theoretical Computer Science 327:32–46. https://doi.org/10.4204/eptcs.327.3

Swords S, Davis J (2011) Bit-blasting ACL2 theorems. In: Proceedings of the 10th International Workshop on the ACL2 Theorem Prover and its Applications, ACL2 2011, Austin, 3–4 Nov 2011, pp 84–102. https://doi.org/10.4204/EPTCS.70.7

Talupur M, Ray S, Erickson J (2015) Transaction flows and executable models: Formalization and analysis of message passing protocols. In: Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, 27–30 Sept 2015, pp 168–175

Temel M, Hunt WA (2021) Sound and automated verification of real-world rtl multipliers. In: 2021 Formal Methods in Computer Aided Design (FMCAD). IEEE, pp 53–62

VL Verilog Toolkit: Documentation. Online (Accessed: 2022). http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____VL

VL Verilog Toolkit. Online (Accessed: 2022). https://github.com/acl2/acl2/tree/master/books/centaur/vl

R.N.M. Watson, P.G. Neumann, J. Woodruff, M. Roe, J. Anderson, D. Chisnall, B. Davis, A. Joannou, B. Laurie, S.W. Moore, others (2016) Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 5). Technical Report UCAM-CL-TR-891. University of Cambridge, Computer Laboratory

Wilding MM, Greve DA, Richards RJ, Hardin DS (2010) Formal verification of partition management for the AAMP7G microprocessor. In: Design and verification of microprocessor systems for high-assurance applications. Springer, Springer, Boston, MA, pp 175–191

Winwood S, Klein G, Andronick J, Elphinstone K, Heiser G, Cock D, Derrin P, Elkaduwe D, Engelhardt K, Kolanski R, Norrish M, Sewell T, Tuch H (2010) seL4: Formal verification of an operating-system kernel. Commun ACM 53(6):107–115

x86isa: Documentation. Online; accessed: 2022. http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____X86ISA

Young WD (1989) A mechanically verified code generator. J Autom Reason 5(4):493–518

Versatile Binary-Level Concolic Testing

38

Bo Chen and Fei Xie

Contents

Introduction 1366
Challenges of Classic Symbolic and Concolic Testing 1367
Overview of Versatile Binary-Level Concolic Testing 1367
Background 1368
Symbolic Execution 1368
Concolic Testing 1369
Related Works 1370
The Infrastructure of Versatile Binary-Level Concolic Testing 1371
Design and Architecture 1372
Real-World Examples 1373
Concolic Testing on COTS Linux Kernel Modules 1375
Design and Architecture 1376
Real-World Examples 1378
Concolic Testing for Hardware/Software Co-validation of Systems-on-Chips 1380
Design and Architecture 1380
Real-World Examples 1383
Conclusions 1385
References 1385

Abstract

Computing systems are experiencing an explosive growth, both in complexities and diversities, ushered in by the proliferation of cloud computing, mobile computing, and Internet of Things. This growth has also exposed the consequences

B. Chen (✉)
Intel Corporation, Hillsboro, OR, USA
e-mail: chen.bo@intel.com

F. Xie
Department of Computer Science, Portland State University, Portland, OR, USA
e-mail: xie@pdx.edu

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_40

of unsafe, insecure, and unreliable computing systems. These all point to the great needs of sophisticated system validation techniques. This chapter presents versatile binary-level concolic testing, which defines a standard execution-trace format, and features an open and highly extensible architecture. It allows easy integration of multiple concrete execution frontends and symbolic execution backends, which significantly improves the applicability and flexibility of symbolic execution, especially to modern computing systems with various components, e.g., operating systems, firmware, and hardware devices. First, this chapter presents the design and implementation of CRETE, the infrastructure of versatile binary-level concolic testing. Second, this chapter presents COD, a framework based on versatile binary-level concolic testing for automated bug detection and replay of commercial off-the-shelf (COTS) Linux kernel modules (LKMs). This framework automatically generates compact sets of test cases for COTS LKMs, proactively checks for common kernel bugs, and allows to reproduce reported bugs repeatedly with actionable test cases. Last, this chapter presents how versatile binary-level concolic testing is leveraged for system-level validation of Systems-on-Chips (SoC). The authors capture runtime traces of hardware/software (HW/SW) components across the entire SoC stack which are emulated by multiple virtual platforms. Based on segmented traces captured from various SoC components, the authors assemble system-level traces and provide interfaces for users to inject system-level assertions to validate.

Keywords

Concolic testing · Symbolic execution · Program analysis · System security · Hardware/software co-validation

Introduction

Computing systems are experiencing an explosive growth, both in complexities and diversities, ushered in by the proliferation of cloud computing, mobile computing, and Internet of Things. This growth has also exposed the consequences of unsafe, insecure, and unreliable computing systems, exemplified by recent high-profile security breaches and software system failures at major corporations such as British Airways (The Guardian 2017) and Facebook (The New York Times 2018). These all point to the great needs of sophisticated system validation techniques. Recent advances in research on symbolic execution (King 1976) have shown great promises for automated software analysis, e.g., generating test cases, finding bugs, and detecting security vulnerabilities (Sen et al. 2005; Godefroid et al. 2005; Song et al. 2008; Cadar et al. 2008; Chipounov et al. 2012; Godefroid et al. 2012; Cha et al. 2012; Cadar and Sen 2013; Kuznetsov et al. 2012; Marinescu and Cadar 2012). However, symbolic execution is mostly adopted to analyze user applications, while modern computing systems in practice consist of many components shipped by various vendors, besides user applications, e.g., operating systems, firmware, and

hardware devices. How to enable symbolic execution on modern computing systems remains a major challenge.

Challenges of Classic Symbolic and Concolic Testing

There have been many recent approaches to symbolic execution (Avgerinos et al. 2014a,b; Shoshitaishvili et al. 2016; Stephens et al. 2016; Redini et al. 2017; Palikareva et al. 2016; Palikareva and Cadar 2013; Bucur et al. 2014; Kasikci et al. 2015; Ramos and Engler 2015; Zheng et al. 2017; Li et al. 2021; Stoenescu et al. 2016). Generally speaking, these approaches can be classified into two categories: online symbolic execution (e.g., BitBlaze (Song et al. 2008), KLEE (Cadar et al. 2008), and S^{2}E (Chipounov et al. 2012)) and concolic execution (a.k.a., offline symbolic execution, e.g., CUTE (Sen et al. 2005), DART (Godefroid et al. 2005), and SAGE (Godefroid et al. 2012)). Online symbolic execution closely couples Symbolic Execution Engines (SEE) with the System Under Test (SUT) and explores all possible execution paths of SUT online at once. On the other hand, concolic execution decouples SEE from the SUT through traces, which concretely runs a single execution path of a SUT and then symbolically executes it. Both online and offline symbolic execution are facing two major challenges for analyzing modern software systems: (1) the SUT involves many types of software for different hardware platforms, and (2) the SUT involves many components distributed on different machines, and as a whole the SUT cannot fit in any SEE.

What's more, modern computing systems consist of many software components from various vendors, and access to all corresponding source code is rarely feasible. Even when source code is available, building the code exactly as in the shipped software product is difficult (Bessey et al. 2010). Moreover, even if the source code is available, compilers can optimize it in many unpredictable ways, such as undefined behaviors in C (Chipounov 2014). Thus, analyses of the software stack of computing systems ought to be at binary level, in order to be practical and useful. Analysis at binary level loses high-level semantics information from the source code that is critical for efficient symbolic analysis. It adds extra complications on top of the two open questions of symbolic execution, namely, state explosion and expensive constraint solving. As a result, optimizations are required to deliver practical techniques that are using symbolic execution.

Overview of Versatile Binary-Level Concolic Testing

Versatile binary-level concolic testing extends classic concolic execution to satisfy the needs for analyzing modern software systems, based on the following two major observations:

- The decoupled architecture of concolic execution provides the flexibility in integrating new trace-captured front ends for modern platforms.

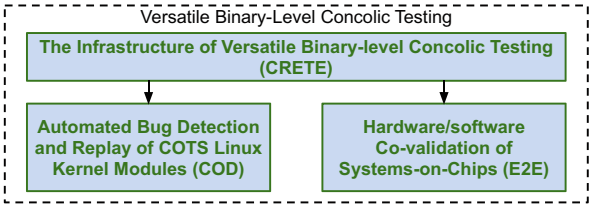


Fig. 1 The overview of versatile binary-level concolic testing

- The trace-based nature of concolic testing offers opportunities for synthesizing system-level traces from different components distributed on different machines.

Versatile binary-level concolic testing features an open and highly extensible architecture allowing easy integration of concrete execution frontends and symbolic execution backends (Chen 2019). Its extensibility is rooted in its modular design where concrete and symbolic execution is loosely coupled only through standardized execution traces and test cases. The standardized execution traces are LLVM-based, self-contained, and composable, providing succinct and sufficient information for SEE to reproduce the concrete executions.

As shown in Fig. 1, the versatile binary-level concolic testing has three major pieces. First, this chapter introduces CRETE, the infrastructure of versatile binary-level concolic testing, which demonstrates its scalability and effectiveness toward real-world applications (Chen et al. 2018). Second, this chapter presents COD, a framework based on versatile binary-level concolic testing for automated bug detection and replay of commercial off-the-shelf (COTS) Linux kernel modules (Chen et al. 2020). Third, this chapter presents how the versatile binary-level concolic testing can be leveraged for hardware/software co-validation of Systems-on-Chips.

Background

This section presents the background of classic symbolic execution and concolic testing.

Symbolic Execution

Symbolic execution (Baldoni et al. 2018) is a program analysis technique that takes symbolic inputs, maintains different execution states and constraints of each path in a program, and utilizes scheduling heuristics (Cha et al. 2018) to effectively explore the execution tree of the target program. An execution state from the symbolic exertion of a program includes a statement counter, values of variables, and a path condition. Since the inputs are symbolic, the values of variables are expressions

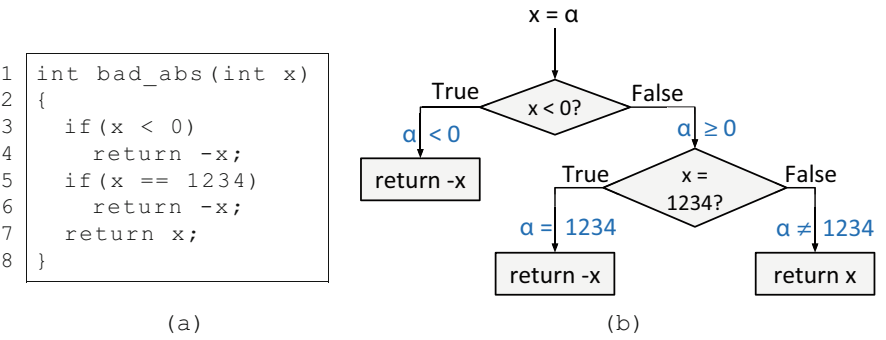


Fig. 2 A simple function `bad_abs` in C with its symbolic execution tree: (**a**) Function `bad_abs` in C. (**b**) Symbolic execution tree of `bad_abs` with symbolic value α assigned to input variable `x`

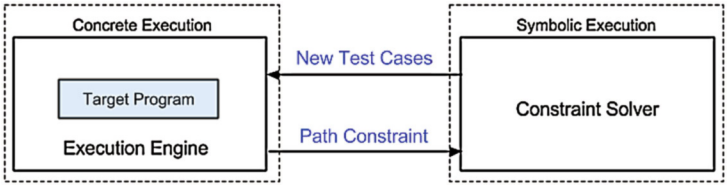


Fig. 3 Concolic testing workflow

over symbolic inputs, and the path condition is a Boolean expression over symbolic inputs. Figure 2 illustrates an example of symbolic execution. At the entry of function `bad_abs`, input `x` is assigned with symbolic value α, which allows all valid values of integer type. For each conditional branch related to symbolic inputs, if both paths are feasible, a new execution state will be forked from the current execution state. By updating path condition based on the branch condition, both paths of the conditional branch can be covered and explored. For this example, symbolic execution forks states twice for two conditional branches, covering three paths in the function.

Concolic Testing

Concolic execution (Sen et al. 2005; Kannavara et al. 2015) combines concrete and symbolic execution. It leverages a concrete execution path to guide symbolic execution to achieve better scalability (Cadar and Sen 2013). It has advantages over concrete execution since it only explores each execution path once based on path constraints, while it is more scalable than symbolic execution because it leverage information from concrete execution to augment symbolic execution. Figure 3 illustrates the basic workflow of concolic testing.

Given an initial test case, the software program under test is concretely executed. During the concrete execution, a trace of the concrete execution is captured, which mainly contains path constraints of the exercised path. By using an offline constraint solver, each branch condition from the captured trace is negated to generate a new test case, aiming at covering new paths of the program under test. Newly generated test cases are fed back into the concrete execution. This process repeats until all paths of the program have been explored or a user specified condition is satisfied.

Related Works

DART (Godefroid et al. 2005) and CUTE (Sen et al. 2005) are both early representative work on concolic testing. They operate on the source code level. CRETE further extends concolic testing and targets close-source binary programs, while it also modularizes concolic testing by loosely coupling concrete execution and symbolic execution only by standardized trace files based on the LLVM bitcode and test cases. SAGE (Godefroid et al. 2012) is a Microsoft internal concolic testing tool that particularly targets at X86 binaries on Windows.

KLEE (Cadar et al. 2008) is a source-code-level symbolic execution tool that is built on the LLVM infrastructure (Lattner and Adve 2004) and is capable of generating high-coverage test cases for C programs. KLEE analyzes the LLVM bitcode compiled from the C SUT, symbolically explores the execution paths of the program, and generates a test case for each path explored. S^{2}E (Chipounov et al. 2012) provides a framework for developing tools for analyzing close-source software programs. It augments a virtual machine (VM) with a SEE and path analyzers. It features a tight coupling of concrete and symbolic execution. The execution of a SUT can cross back and forth between concrete and symbolic execution.

BitBlaze (Song et al. 2008) is an early representative work on binary analysis for computer security. It provides TEMU, a QEMU-based runtime analysis frontend, and VINE, a symbolic execution backend. TEMU and VINE were closely integrated into Rudder, a tool for symbolic execution of software binaries. BitBlaze, particularly Rudder, focuses on effective detection of security vulnerabilities by leveraging the close coupling of TEMU and VINE. Mayhem (Cha et al. 2012) and MergePoint (Avgerinos et al. 2014a) build on BitBlaze and further optimize the close coupling of their concrete execution frontend and symbolic analysis backend to improve their effectiveness in detecting exploitable software bugs.

ANGR is an extensible Python framework for binary analysis using VEX (Nethercote and Seward 2007) as an intermediate representation (IR). It implemented a number of existing analysis techniques and enabled the comparison of different techniques in a single platform. ANGR provides CLE to load the binary under test in its own virtual environment and provides lifters to disassemble binary code into VEX IR, from where it conducts symbolic execution over VEX IRs. As ANGR performs in vitro binary analysis, it requires to model the real execution environment for the binary under test, like system calls and common library functions. This is

one of the biggest limitations of angr, because the environment model can never be complete nor accurate.

Much research has been done in the area of HW/SW co-validation which are close to CRETE. HW/SW co-verification is a common technique which mainly uses model checking to verify HW/SW interface protocols against the driver and various device models (Kurshan et al. 2002; Mukherjee et al. 2017; Corteggiani et al. 2021; Jakobs et al. 2021; Lyu and Mishra 2021). Recently research work leverages virtual devices for HW/SW co-validation and SoC validation (Gu et al. 2018; Lei et al. 2019). Symbolic execution with VD co-verification is proposed to verify hardware and firmware interactions (Horn et al. 2013; Alam et al. 2022). This work either focuses on device/driver interfaces or device/firmware interfaces, which lacks holistic system-level analysis of the entire SoC stack.

The Infrastructure of Versatile Binary-Level Concolic Testing

The CRETE framework for binary-level concolic testing features several key design goals:

- **Binary-Level In Vivo Analysis.** It requires only the binary of the SUT and performs analysis in its real execution environment.
- **Extensibility.** It allows easy integration of concrete execution frontends and SEE backends.
- **High Coverage.** It achieves coverage that is not significantly lower than the coverage attainable by source-level analysis.
- **Minimal Changes to Existing Testing Processes.** It should simply provide additional test cases that can be plugged into existing testing processes without major changes to the testing processes.

To achieve the goals above, the CRETE framework adopts an online/offline approach to concolic testing in its design:

- **Online Tracing.** As the SUT is concretely executed in a virtual or physical machine, an online tracing plugin captures the binary-level execution trace into a trace file.
- **Offline Test Generation.** An offline SEE takes the trace as input, injects symbolic values, and generates test cases. The new test cases are in turn applied to the SUT in the concrete execution.

This online tracing and offline test generation process is iterative: it repeats until all generated test cases are issued or time bounds are reached. The CRETE framework extends this process to satisfy its design goals as follows.

- Execution traces of a SUT are captured in its unmodified execution environment on binary level. The tracing plugin can be an extension into a VM, a hardware

tracing facility, or a dynamic binary instrumentation tool, such as PIN (Luk et al. 2005) and DynamoRIO (Bruening et al. 2012).
- The concrete and symbolic execution environments are decoupled by standardized traces. As long as they can generate and consume standardized traces, they can work together as a cohesive concolic process.
- Optimization can be explored on both tracing and test case generation, for example, selective binary-level tracing to improve scalability and concolic test generation to reduce test case redundancy. This makes high-coverage test generation on binary level possible.
- The tracing plugin is transparent to existing testing processes, as it only collects information. Therefore, no change is made to the testing processes.

Design and Architecture

As shown in Fig. 4, CRETE has four key components: **CRETE Runner**, a tiny helper program executing in the guest OS of the VM, which parses the configuration file and launches the target binary program (TBP) with the configuration and test cases; **CRETE Tracer**, a comprehensive tracing plugin in the VM, which captures binary-level traces from the concrete execution of the TBP in the VM; **CRETE Replayer**, an extension of the SEE, which enables the SEE to perform concolic execution on the captured traces and to generate test cases; **CRETE Manager**, a coordinator that integrates the VM and SEE, which manages runtime traces captured and test cases generated, coordinates the concrete and symbolic execution in the VM and the SEE, and iteratively explores the TBP.

CRETE takes a TBP and a configuration file as inputs, and outputs generated test cases along with a report of detected bugs. The manual effort and learning curve to utilize CRETE are minimal. It makes virtually no difference for users to set up the testing environment for the TBP in a CRETE-instrumented VM than a vanilla VM. The configuration file is an interface for users to configure parameters on testing a TBP, especially specifying the number and size of symbolic command-line inputs and symbolic files for test case generation.

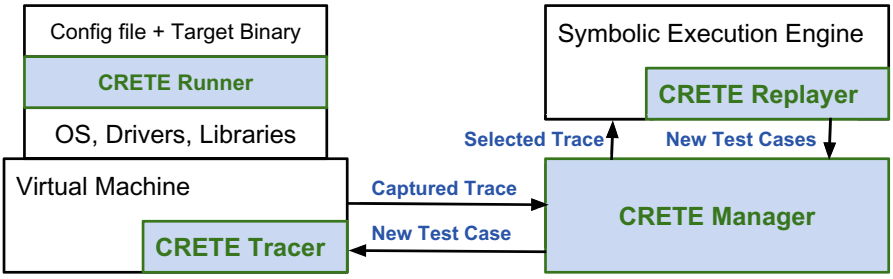


Fig. 4 CRETE architecture

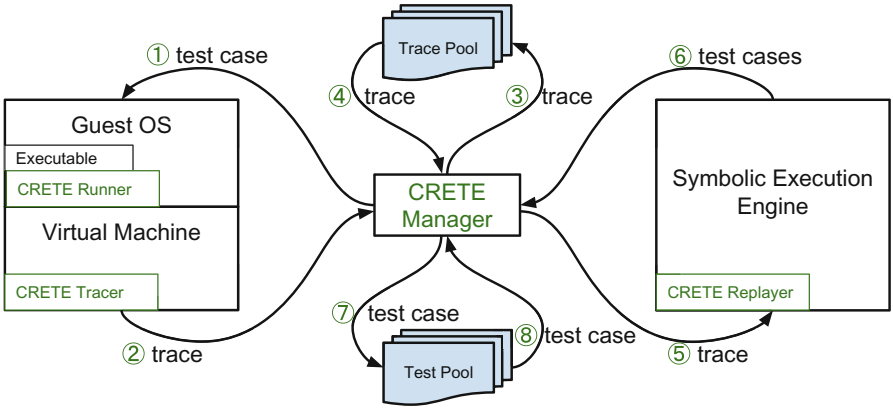


Fig. 5 CRETE workflow

The workflow of CRETE is shown in Fig. 5. CRETE works in iterations and each iteration includes the following phases:

- **Binary Execution Phase:** CRETE Runner first loads the input binary and a test case into the guest OS. Then CRETE Runner executes the binary with the data defined in the test case as inputs. In this way, the binary is executed within VM in its native, unmodified guest OS environment.
- **Trace Capture Phase:** Along with the execution of the target program, CRETE Tracer captures the runtime information needed to constitute a runtime trace for symbolic analysis.
- **Trace Selection Phase:** CRETE Manager takes the captured trace as input and maintains a pool of traces. CRETE Manager then selects a trace from this pool and passes it to CRETE Replayer.
- **Offline Replaying Phase:** CRETE Replayer, in turn, invokes the SEE to execute the selected trace symbolically. The SEE performs concolic test case generation.
- **Test Selection Phase:** CRETE Manager receives newly generated test cases from the SEE and maintains a test case pool. CRETE Manager then selects one test case from the pool and sends it back to CRETE Runner to start the next iteration of CRETE. This workflow iterates until no more test cases can be generated or user-specified time bounds are reached.

Real-World Examples

The authors present the applications of CRETE to GNU COREUTILS (GNU) and TianoCore utility programs for UEFI BIOS (Tianocore). Those evaluations demonstrate that CRETE generates effective test cases that are as effective in achieving high code coverage as the state-of-the-art tools for automated test case generation and can detect serious deeply embedded bugs.

GNU COREUTILS is a package of utilities widely used in Unix-like systems. It is an often used benchmark for evaluating automated program analysis systems, including KLEE, MergePoint, and others (Cadar et al. 2008; Wong et al. 2015; Avgerinos et al. 2014a). As shown in Table 1, the experiments demonstrate that CRETE achieves comparable test coverage to KLEE and generally outperforms ANGR. Note CRETE and ANGR generate test cases from program binaries without debug information, while KLEE requires program source code. The major advantage of KLEE over CRETE is that it works on source code with all semantics information available. When the program size is small, symbolic execution is capable of exploring all feasible paths with given resources, such as time and memory. This is why KLEE can achieve great code coverage, such as line coverage over 90%, on more programs than CRETE, as shown in Table 2. KLEE requires to maintain execution states for all paths being explored at once. This limitation becomes bigger when size of program gets bigger. What's more, KLEE analyzes programs within its own virtual environment with simplified model of real execution environment. Those models sometimes offer advantages to KLEE by reducing the complexity of the TBP, while sometimes they lead to disadvantages by introducing inaccurate environment. This is why CRETE gradually caught up in general as shown in Table 2. Specifically, CRETE gets higher line coverage on 33 programs, lower on 31 programs, and the same on other 23 programs.

TianoCore utility programs are part of the open-source project EDK2 (Tianocore), a cross-platform firmware development environment from Intel. It includes 16 command-line programs used to build BIOS images. Figure 6 shows that CRETE

Table 1 Comparison of average and median coverage by KLEE, ANGR, and CRETE on COREUTILS

	Line (%)			Function (%)			Branch (%)		
Cov.	KLEE	ANGR	CRETE	KLEE	ANGR	CRETE	KLEE	ANGR	CRETE
Average.	70.48	66.79	74.32	78.54	79.05	83.00	58.23	54.26	63.18
Median.	88.09	81.62	86.60	100	100	100	79.31	70.59	77.57

Table 2 Distribution comparison of coverage achieved by KLEE, ANGR, and CRETE on COREUTILS

	Line			Function			Branch		
Cov.	KLEE	ANGR	CRETE	KLEE	ANGR	CRETE	KLEE	ANGR	CRETE
90–100%	40	24	33	65	60	65	15	16	19
80–90%	15	22	25	12	8	10	27	12	17
70–80%	13	14	10	3	7	5	14	16	25
60–70%	9	12	10	2	4	3	9	15	6
50–60%	5	7	4	1	4	1	8	11	9
40–50%	1	2	3	1	1	2	8	7	6
0–40%	4	6	2	3	3	1	6	10	5

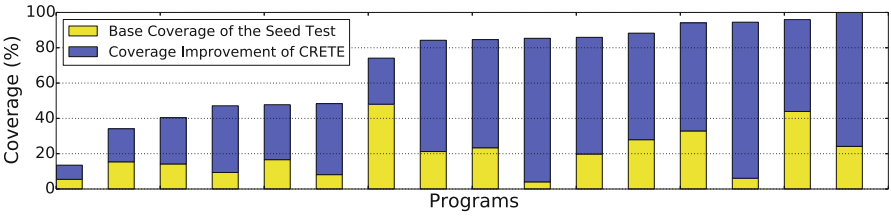


Fig. 6 High coverage from Scratch by CRETE on TianoCore utilities

Table 3 Classified crashes found by CRETE on TianoCore utilities: 84 unique crashes from eight programs

Crash type	Count	severity	Crashed programs
Stack corruption	1	High (exploitable)	VfrCompile
Heap error	6	High (exploitable	GenFw
Write access violation	23	High (exploitable)	EfiLdrImage, GenFw, EfiRom, GenFfs
Abort signal	2	Medium (signs of exploitable)	GenFw
Read access violation	45	Low (may not be exploitable)	GenSec, GenFw, Split, GenCrc32, VfrCompile
Other access violation	7	Mixed	GenFw

delivered high code coverage from scratch, above 80% line coverage, on 9 out of 16 programs. What's more, CRETE found 84 distinct crashes (by stack hash) from eight TianoCore utility programs. Table 3 shows that CRETE found various kinds of crashes including many exploitable ones, such as stack corruption, heap error, and write access violation. Most of the crashes were confirmed as real bugs, and ten of them were fixed promptly in the upstream.

Concolic Testing on COTS Linux Kernel Modules

Linux kernel is pervasive in the cloud, on mobile platforms, and on supercomputers. To support these diverse computing environments, the Linux kernel provides extensibility and modularity through Loadable Kernel Modules (LKM) while featuring a monolithic architecture for execution efficiency. This architecture design brings a major challenge to the security of Linux kernel. Having LKMs run in the same memory space as the base kernel on Ring 0, a single flaw from LKMs may compromise the entire system, e.g., gaining root access. However, validation and debugging of LKMs are inherently challenging, because of its special interface buried deeply in the kernel, @and non-determinism from interrupts. Also, LKMs are shipped by various vendors and may not have access to their source code, making the validation even harder.

This section presents COD, a framework for efficient bug detection and replay of commercial off-the-shelf (COTS) Linux kernel modules based on the versatile binary-level concolic testing. The framework automatically generates compact sets of test cases for COTS LKMs, proactively checks for common kernel bugs, and allows to reproduce reported bugs repeatedly with actionable test cases.

Design and Architecture

The COD framework features the following design goals for analyzing LKMs:

- **Binary-Level In Vivo Analysis.** It is applicable to COTS LKMs and require nos recompilation or modification to the rest of kernel stack.
- **Effective Bug Detection.** It detects various types of kernel bugs with minimal false alarms.
- **Automated Bug Replay.** It enables developers to reproduce bugs easily, which helps locate and fix the reported bugs.
- **Multiple LKMs.** It is capable of analyzing multiple LKMs and their interactions at the same time.

To achieve the goals above, the versatile binary-level concolic testing approach of CRETE is adopted and extended in the design of the COD framework as follows.

- The COD framework introduces a `kernel shim` to intercept interactions between base kernel and target LKMs and uses it along with a `kernel hypercall interface` to dynamically inject concolic values at LKM interfaces while capturing runtime traces. Also, it builds `COD tracer` by augmenting CRETE tracer to support multiple applications and kernel modules, through which COD captures runtime execution traces of target LKMs from unmodified guest OS stack.
- The COD framework builds `COD Trace Replayer` for symbolic analysis and test case generation over the captured traces, by extending the CRETE trace replayer with `trace checkers` and `constraint editors` for checking common kernel bugs and imposing constraints on generated test cases.
- The COD framework provides `COD TC Replayer`, which allows users to replay generated test cases repeatedly, out of test generation environment and on both virtual and physical platforms. It is embedded with `kAPI checkers` to detect common kernel bugs and produce informative reports to boost bug analysis.

As shown in Fig. 7, the COD architecture for test case generation is split into two domains, VM guest OS and host OS. A user-land `Agent` and two custom kernel modules, `kernel shim` and `kernel hypercall interface`, together with target LKMs and native OS stack are running within VM guest OS. A virtual

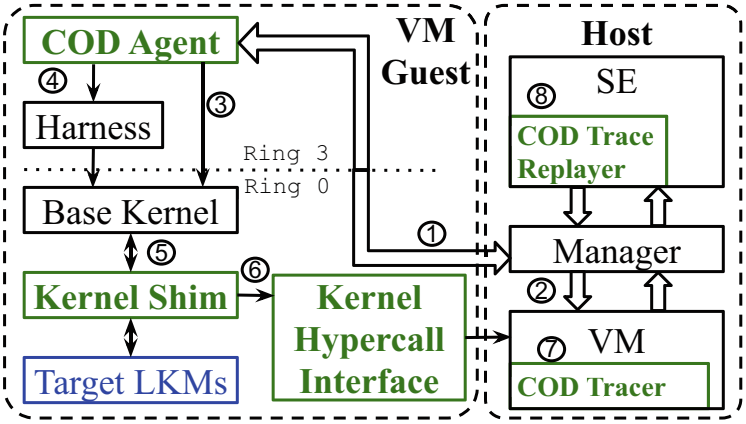


Fig. 7 The architecture of COD for automated test case generation

machine augmented with `COD Tracer`, a symbolic engine augmented with `COD Trace Replayer`, and a Manager are running on host machine.

Here is the events and communications that take place during the test case generation process of the COD framework. When the manager is started, (1) it sends a message to `Agent` through sockets and (2) sends an initial test case to the VM. The message contains a list of target LKMs and a sequence of commands as test harness. (3) The `Agent` loads two custom kernel modules, `kernel shim` and `kernel hypercall interface`, and passes them the list of LKMs as parameters. (4) The `Agent` then executes the commands of the test harness sequentially to trigger functionalities of target LKMs through base kernel. (5) The custom kernel module `kernel shim` intercepts the interactions between base kernel and target LKMs. (6) It also communicates with the VM through the other module `kernel hypercall interface`, to add new tainted values to the taint analysis engine in the VM, report kernel panics to the VM, and retrieve values of test case from VM to modify the interactions between target LKMs and base kernel if needed. (7) When all commands in the test harness are finished, the `COD Tracer` captures the runtime execution trace into a file and sends it to symbolic engine through the manager over sockets. (8) The `COD Trace Replayer` performs symbolic analysis over the captured trace and sends the generated test cases back to the VM. The iteration of test case generation repeats from step (4) to step (8) and stops when user-specified conditions are met, e.g., time limits.

COD allows user to reproduce generated test cases repeatedly on both physical and virtual machines and generates crash log to assist developers to debug and fix reported bugs. As shown in Fig. 8, the architecture of test case replay in COD is composed of a user-mode program `TC Replayer` with an extensible plugin `kAPI Checker` and three custom kernel modules, namely, `Kernel Shim`, `TC Element Supplier`, and `kAPI Tracer`. The workflow of this design

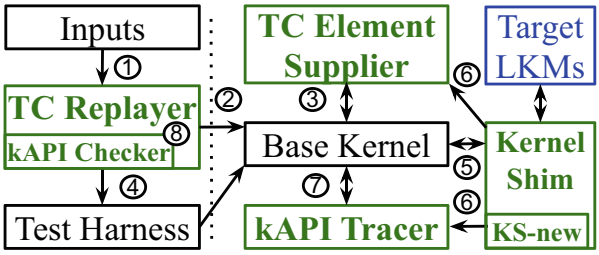


Fig. 8 The architecture of COD for automated test case replay

is illustrated below. (1) The `TC Replayer` is started by users with inputs of a set of test cases and a configuration file. The configuration file contains a list of target LKMs and a sequence of commands as the test harness. Then the `TC Replayer` (2) loads the three custom kernel modules and passes them the list of target LKMs as parameters, (3) picks one test case and passes it to the custom kernel module `TC Element Supplier`, and (4) executes the commands in the test harness sequentially to trigger functionalities of target LKMs. (5) The custom kernel module `Kernel Shim` intercepts the interactions between base kernel and target LKMs. (6) The callbacks in `Kernel Shim` either call into `TC Element Supplier` to modify the interactions between kernel and target LKMs or call into `kAPI Tracer` to capture kernel API usage information. When all commands in the test harness are executed, the `TC Replayer` (7) retrieves the kernel API usage information from the custom kernel module `kAPI Tracer` and (8) checks for potential bugs with `kAPI Checker`. The loop repeats from (3) to (8) for all input test cases.

Real-World Examples

The effectiveness of the COD framework is highlighted by its applications to LKMs that are widely used and validated both in the industry and academia. Table 4 shows the list of LKMs that were evaluated with COD. All the main LKMs have been released over 15 years (Torvalds 2005). They are also being actively maintained by the Linux kernel community and large vendors, such as RedHat, SUSE, Broadcom, and Intel. This is because those LKMs are providing important functionality to modern computer systems, such as Ethernet device drivers, network middleware, HDA codec, core sound module, etc. For the same reason, many of the LKMs, e.g., `E1000`, `PCNet32`, `8139too`, and `snd_ens1370`, have been studied and used as benchmarks for evaluation by numerous previous research prototypes (Renzelmann et al. 2012; Cong et al. 2015; Bai et al. 2016).

The COD framework was applied for test generation with a time-out of 24 h on each main LKM along with their dependent LKMs as listed in Table 4. By replaying all generated test cases with COD on both virtual and physical machines, COD

Table 4 List of LKMs Evaluated by COD

Subsystem	Main LKM	Dependent LKMs
Network	e100	mii
	e1000	–
	pcnet32	mii
	ne2k-pci	8390
	8139too(cp)	mii
	tg3	ptp, pps_core
Sound	snd_intel8x0	snd-ac97-codec, ac97_bus, snd-pcm, snd-timer, snd, soundcore
	snd_hda_intel	snd_hda_codec_generic,snd_hda_codec, snd_hda_core, snd_hwdep,snd_pcm, snd_timer, snd, soundcore
	snd_ens1370	snd_rawmidi, snd_seq_devicesnd_pcm, snd_timer, snd, soundcore

Table 5 New Linux kernel vulnerabilities detected by COD

Index	LKM	Bug description	Patch hash
1	E1000	Resource leak	`ee400a3`
2	E1000	Null-pointer dereference	`cf1acec`
3	Pcnet32	Resource leak	`d7db318`
4	8139too(cp)	Kernel API misuse	`a456757`
5	hda_intel	Null-pointer dereference	`a3aa60d`

reported a total of five new distinct vulnerabilities from four different kernel module. As shown in Table 5, COD detected various kinds of vulnerabilities, including null-pointer dereference, resource leak, and kernel API misuse. All the bugs were reported to the Linux kernel community and were patched immediately. The links of the submitted bugs are omitted for double-blind review purpose.

The authors take `Bug 1` as an example to explain why COD is able to generate test cases from COTS LKMs to trigger and report the new flaws in Table 5. `Bug 1` is detected by `TC Replayer` during the replay of COD generated test cases, where `kAPI checker` reported a piece of memory allocated by function `__kmalloc` is not paired with any memory de-allocation function. By examining test cases triggering this bug, the authors found COD only flipped a single kernel API return from the initial test case. COD was able to explicitly flip these single API returns because there are conditional branches in the target LKM depending on the flipped API returns. By leveraging concolic execution, COD was able to negate these branch conditions precisely, generate a compact set of test cases to explore new code in the LKM, and finally catch the bug with `TC Replayer` and `kAPI checker`. For the similar reason, COD flipped more kernel APIs, generated LKM test cases with the right kernel API combination to reach error paths, and finally reported these vulnerabilities with `TC Replayer`.

Concolic Testing for Hardware/Software Co-validation of Systems-on-Chips

Many approaches have been proposed to improve the quality of Systems-on-Chips (SoC), mainly focusing on a specific part of the SoC, e.g., device driver, hardware, firmware, etc. System-level validation of the entire SoC stack remains a major challenge, and so far research on end-to-end validation of SoC that covers both hardware and software (HW/SW) components is comparatively sparse. This section presents the framework to end-to-end concolic testing for HW/SW co-validation of SoC (Chen et al. 2019) leveraging versatile binary-level concolic testing. Based on the simulation of SoC with multiple virtual platforms, the framework captures a set of runtime traces from different components of the entire SoC and assembles them into holistic system-level traces. It also provides instrumentation interfaces over the SoC trace for custom validation and analysis, allowing insertions of user-defined assertions and symbolic values at various HW/SW interfaces. The instrumented trace is replayed in a concolic/symbolic engine to generate new system-level test cases that either explore new paths of the SoC stack or trigger assertions.

Design and Architecture

As shown in Fig. 9, the framework mainly has two phases, online tracing and offline analysis. At runtime (online), it performs end-to-end tracing over the entire SoC stack emulated by multiple virtual platforms (VP), from which a sequence of traces is captured, including host SW traces, virtual device (VD) traces, and firmware traces. Statically (offline), the framework assembles segmented traces into a holistic system-level trace, provides instrumentation interfaces for user-defined assertions and symbolic values over the assembled trace, and utilizes concolic/symbolic engines to generate test cases that either explore new usages of the SoC or trigger user-defined assertions.

A set of tracers are provided to each SoC hardware component for runtime tracing, as shown in Fig. 9. The tracer for host SW is an extension to the SoC host VP. When a target application is invoked, the tracer captures user inputs as r^h and takes a snapshot of the VP's CPU and memory as s^h. It also monitors the complete execution of host SW to capture a sequence of machine-level instructions as the execution path of host SW π^h. The VD tracer is a wrapper to the IP VD, which intercepts all interactions between the IP VD and the SoC host VP. For each host SW/VD interaction, the tracer captures the VD requests as r^v and takes the snapshot of the VD state as s^v. The π^v is a concrete execution path of the VD and can be derived from the VD source code with the captured r^v. The tracer for firmware is an extension to the IP Core VP. For each request from VD, it captures the request input as r^f, takes a snapshot of the VP's CPU and memory before the execution of the firmware as s^f, and monitors the complete execution of the firmware to capture a sequence of machine-level instructions as the execution path of firmware π^f. A unified instruction format is needed to make traces captured by different tracers compatible.

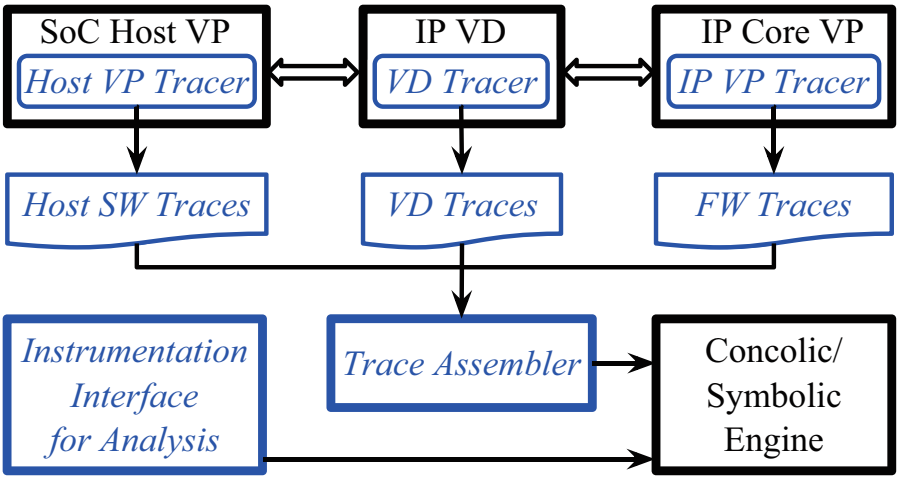


Fig. 9 Architecture and workflow of end-to-end concolic testing for hardware/software co-validation: (1) execute SoC software stack over different VPs with partitioned VDs; (2) capture segmented traces from UOD, VD, and firmware, respectively; (3) assemble a system-level trace and inject system- level assertions; and (4) inject symbolic values at HW/SW interfaces and perform concolic-symbolic hybrid execution, generating test cases to cover new usage of the SoC or trigger assertions

As shown in Algorithm 1, the system-level trace assembler takes $\tau^h, \mathcal{T}^v, \mathcal{T}^f$ as inputs, where τ^h is the captured host SW trace, $\mathcal{T}^v$ is a sequence of captured VD traces, and $\mathcal{T}^f$ is a sequence of captured firmware traces. The *project* operator STATE takes a trace $\langle r, s, \pi \rangle$ and returns the state element s. The check on whether an instruction $i \in \pi$ is a special instruction or not is implementation-dependent, which is related to the instruction set of the target VP and how it emulates its interaction with VDs. Function APPEND (x, y) appends element y to the end of sequence x. Function NEXT (x) returns the first element in the sequence x and removes the element from x. Special `NOPs` in the execution path of the system-level trace include `NopEnterVd`, `NopLeaveVd`, `NopEnterFw`, and `NopLeaveFw` to assist the execution transition between host software and VD or between VD and firmware. For example, `NopEnterVd` correlates a special host SW instruction with a VD stimulus r^v, synchronizes the VD state with s^v, and transfers execution to VD, while `NopLeaveVd` propagates VD's return value back to s^h and transfers execution back to host SW. `NopEnterFw` and `NopLeaveFw` provide similar functionalities at the VD/FW interface.

Algorithm 2 shows the algorithm to interpret the system-level trace τ^S. The interpreter iterates through all the instructions in the execution path π^S and executes them in sequence. When `NopEnterVd` and `NopEnterFw` are encountered, the framework synchronizes the VD and firmware states from the captured states accordingly. The *callbacks* defined as the second argument in algorithm 2 serves as an instrumentation interface to allow the user to easily inject custom functionalities and checks. The user-defined callback functions are written in high-level

Algorithm 1: ASSEMBLE-SYS-TRACE $\left(\tau^h, \mathcal{T}^v, \mathcal{T}^f\right)$

```
1  ⟨r^h, s^h, π^h⟩ ← τ^h
2  S^v ← map (STATE, T^v) , S^f ← map (STATE, T^f)
3  π ← []                                   ▷ initialize π to be an empty sequence
4  foreach i^h ∈ π^h do
5      if i^h is normal instruction then APPEND(π, i^h)
6      else                                 ▷ i^h interacts with virtual device
7          APPEND(π, NopEnterVd)
8          ⟨r^v, s^v, π^v⟩ ← NEXT(T^v)
9          foreach i^v ∈ π^v do
10             if i^v is normal instruction then APPEND(π, i^v)
11             else                         ▷ i^v interacts with firmware
12                 APPEND(π, NopEnterFw)
13                 ⟨r^f, s^f, π^f⟩ ← NEXT(T^f)
14                 foreach i^f ∈ π^f do
15                     APPEND(π, i^f)
16                 APPEND(π, NopLeaveFw)
17         APPEND(π, NopLeaveVd)
18 return ⟨r^h, s^h, S^v, S^f, π⟩
```

Algorithm 2: INTERPRET-SYS-TRACE $\left(\tau^S, callbacks\right)$

```
1  ⟨r^h, s^h, S^v, S^f, π^S⟩ ← τ^S
2  foreach i ∈ π^S do
3      switch i do
4          case NopEnterVd : do s^v ← NEXT(S^v)
5          case NopEnterFw : do s^f ← NEXT(S^f)
6          otherwise do EXECUTE(⟨s^h, s^v, s^f⟩, i)
7      if i is Nops then
8          PROCESS-CALLBACKS(⟨r^h, s^h, s^v, s^f⟩, callbacks)
```

programming languages, such as *C*. Those callback functions will be called at corresponding special `NOPs`. User-defined functions have access to various runtime states in the SoC trace τ^S, namely, r^h, s^h, s^v, and s^f, and the information from HW/SW interactions, e.g., value writing from the IP driver to VD interface registers. In this way, users can check properties related to the state of host SW, VD, and FW, as well as properties related to HW/SW interactions. For example, users can put an assertion to check whether user inputs to IP applications from host SW can cut across the entire SoC HW/SW stack and directly control the execution state of IP firmware. Also, users can introduce symbolic values at different levels of SoC stack, mainly at the HW/SW interfaces. This enables more thorough exercise of the captured SoC trace, making concolic/symbolic hybrid execution possible. This also allows users to make trade-offs between soundness and completeness over

analysis of the SoC trace, which is similar to various consistency models described in S2E (Chipounov et al. 2012).

Real-World Examples

The usefulness and effectiveness of the framework is demonstrated by its applications to validating an Ethernet IP, Intel E1000 gigabyte network adapter (Chen et al. 2019). In its experiment, eight assertions were defined to enforce eight system-level properties that the target SoC should hold. These properties are retrieved from the specification of Intel E1000 network adapter (Intel 2009). As shown in Table 6, these properties are related to the Receive Control Register (`RCTL`) and Control Register (`CTRL`) of E1000. Figure 10 shows three examples of the defined assertions, which illustrates how the instrumentation interface of the framework is used to express and inject the assertions into the system-level traces. The assertions in the examples are mainly related to the interface register `RCTL`. For example, assertions $P1$ and $P2$ check properties related to MMIO at the interface of SW driver and VD. Assertions can also check on the entire SoC states, such as assertion $P3$ is based on both the VD state and user inputs to stimulus (network applications). The experiment also injected symbolic values at host SW/VD and VD/FW interfaces, as well as the user inputs to stimulus. Figure 10 shows an example of injecting symbolic values to `value` which are the input of VD interface function passed from the IP driver. Then the instrumented traces are applied to the symbolic/concolic engine for system-level analysis and validation. Table 7 shows the experiment results over the trace captured from stimulus #1. It contains the number of test cases being generated, the number of assertions being validated, the number of assertions being violated, and the number of actual bugs being detected.

Table 6 System-level properties validated by user-defined assertions

#	Description
1	The software should only write `0x00`/`0x01` to `RCTL.LBM`
2	When `RCTL.BSEX` is `0x01`, the software should not program value `0x00` to `RCTL.BSIZE`
3	The values `0x10` and `0x11` are reserved to `RCTL.DTYP`
4	When `RCTL.FLXBUF` is not `0x00`, the receive buffer size is represented by the value of `RCTL.FLXBUF` in KB
5	When `RCTL.DTYP` is `0x01`, the buffer sizes for the descriptor are controlled by fields in the `PSRCTL` register
6	The first byte of `RCTL` is reserved
7	When a write to `CTRL` is finished, `CTRL.RST` should always be cleared by the device every time
8	The value `0x11` is reserved to `CTRL.SPEED`

```
void pre_vd_mmio_op(uint32_t *reg_idx,
                    uint32_t *value, int is_write) {
  // Introduce symbolic values to VD input 'value'
  make_symbolic(value, *value, sizeof(*value));

  if(reg_idx == RCTL && is_write) {
    //P1: Only 00/01'b is valid to LBM
    uint8_t lbm = (*value >> LBM) & 0x03;
    assert( lbm == 0x00 || lbm == 0x01);
    //P2: 00'b is invalid to DTYP when BSEX is 01'b
    uint8_t bsex = (*value >> BSEX) & 0x01;
    uint8_t bsize = (*value >> BSIZE) & 0x03;
    if(bsex == 0x01)
      assert(bsize != 0x00);
  }
}
void post_vd_mmio_op(uint32_t reg_idx,
                     uint32_t value, int is_write) {
  if(reg_idx == RCTL && is_write) {
    //P3: when FLXBUF is not 00'b, use FLXBUF
    //       in KB as the rx buffer size
    uint8_t flxbuf = (value >> FLXBUF) & 0x0f;
    if(flxbuf != 0x00)
      assert(vd_state.rxbufsize == (flxbuf * 1024));
  }
}
```

Fig. 10 User-defined assertion examples in pseudo C code

Table 7 Number of generated test cases and triggered assertions from concolic-symbolic hybrid execution

	User-inputs to stimulus	Driver/VD interface	VD/firmware interface
Generated test cases	20	1001	49
Validated assertions	5	1	4
Fired assertions	1	7	2
Detected bugs	1	3	1

Symbolic values of user inputs to stimulus generated the least number of test cases and triggered the least number of assertions since they crosscut the entire SoC stack and accumulate complete constraints of the SoC execution. Following the strictest constraints also makes all generated test cases valid to the entire SoC and hence does not introduce false alarms. Behind the only assertion failure triggered by the test cases of application inputs, a bug in the FW is discovered. Although this bug is handcrafted, it demonstrates that the framework can precisely explore the impact of user inputs to the top level of the host SW stack across the entire SoC HW/SW stack. It generated an exact test case that pinpoints the FW buggy path from the user inputs to the host SW.

Symbolic values of driver/VD interfaces generated the largest number of test cases and triggered the largest number of assertions, while it has a much higher false rate on assertions triggered. By following partial constraints of the SoC stack, it is

easier to explore partial stack more thoroughly, but it also produces test cases that might be invalid to the entire SoC stack. Manual efforts are needed to review all the triggered assertions. In the experiment, there are *seven* triggered assertions in total where *three* of them are real alarms and report real bugs. Besides the handcrafted FW bug, two bugs from the E1000 VD in QEMU are detected by the approach. One is reported by assertion $P3$ as shown in Fig. 10, and both of them are the functionalities that are required according to the Intel E1000 Manual while not being implemented in QEMU's E1000 VD. Moreover, as the FW is written by us and has basic logic, the test cases and triggered assertions from VD/FW interface are much less compared to those from driver/VD interface.

Conclusions

This chapter introduced versatile binary-level concolic testing, which significantly improves the applicability and flexibility of symbolic execution, especially to modern computing systems with various components. The first part of the chapter presented CRETE, the infrastructure of versatile binary-level concolic testing, to enable symbolic execution on modern computing systems, and scale it with a set of optimizations, which delivered competitive results comparing with state-of-theart tools for automated software analysis and detected numerous unknown bugs from various real-world applications. Second, this chapter introduced the design and architecture of COD, a system for automated bug detection and replay of COTS Linux kernel modules, which makes the Linux kernel more reliable and secure by detecting and fixing various unknown vulnerabilities. At last, this chapter presented an approach to HW/SW co-validation with end-to-end concolic testing, which helps tackle the challenge of system-level validation over the entire SoC stack.

References

Alam T, Yang Z, Chen B, Armour N, Ray S (2022) Firver: concolic testing for systematic validation of firmware binaries. In: 27th Asia and South Pacific design automation conference, ASP-DAC 2022, Taipei, 17–20 Jan 2022. IEEE, pp 352–357

Avgerinos T, Rebert A, Cha SK, Brumley D (2014a) Enhancing symbolic execution with veritesting. In: 36th international conference on software engineering, ICSE'14, Hyderabad, pp 1083–1094

Avgerinos T, Cha SK, Rebert A, Schwartz EJ, Woo M, Brumley D (2014b) Automatic exploit generation. Commun ACM 57(2):74–84

Bai JJ, Wang YP, Yin J, Hu SM (2016) Testing error handling code in device drivers using characteristic fault injection. In: Proceedings of the 2016 USENIX conference on Usenix annual technical conference, USENIX ATC'16, Berkeley. USENIX Association, pp 635–647

Baldoni R, Coppa E, D'Elia DC, Demetrescu C, Finocchi I (2018) A survey of symbolic execution techniques. ACM Comput Surv 51(3):50:1–50:39

Bessey A, Block K, Chelf B, Chou A, Fulton B, Hallem S, Henri-Gros C, Kamsky A, McPeak S, Engler D (2010) A few billion lines of code later: using static analysis to find bugs in the real world. Commun ACM 53(2):66–75

Bruening D, Zhao Q, Amarasinghe S (2012) Transparent dynamic instrumentation. In: Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on virtual execution environments, VEE'12, New York. ACM, pp 133–144

Bucur S, Kinder J, Candea G (2014) Prototyping symbolic execution engines for interpreted languages. In: Proceedings of the 19th international conference on architectural support for programming languages and operating systems, ASPLOS'14, New York. ACM, pp 239–254

Cadar C, Sen K (2013) Symbolic execution for software testing: three decades later. Commun ACM 56(2):82–90

Cadar C, Dunbar D, Engler D (2008) Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX conference on operating systems design and implementation, OSDI'08, pp 209–224

Cha SK, Avgerinos T, Rebert A, Brumley D (2012) Unleashing mayhem on binary code. In: IEEE symposium on security and privacy, SP2012, San Francisco, pp 380–394

Cha S, Hong S, Lee J, Oh H (2018) Automatically generating search heuristics for concolic testing. In: Proceedings of the 40th international conference on software engineering, ICSE 2018, Gothenburg, pp 1244–1254

Chen B (2019) Versatile binary-level concolic testing. PhD thesis, Portland State University

Chen B, Havlicek C, Yang Z, Cong K, Kannavara R, Xie F (2018) CRETE: a versatile binary-level concolic testing framework. In: Fundamental approaches to software engineering. Springer International Publishing, Cham, pp 281–298

Chen B, Cong K, Yang Z, Wang Q, Wang J, Lei L, Xie F (2019) End-to-end concolic testing for hardware/software co-validation. In: Proceedings of the 15th IEEE international conference on embedded software and systems, ICESS 2019, Las Vegas

Chen B, Yang Z, Lei L, Cong K, Xie F (2020) Automated bug detection and replay for COTS linux kernel modules with concolic execution. In: 27th IEEE international conference on software analysis, evolution and reengineering, SANER 2020, London, 18–21 Feb 2020. IEEE, pp 172–183

Chipounov V (2014) S2E: a platform for in-vivo multi-path analysis of software systems. PhD thesis, ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Chipounov V, Kuznetsov V, Candea G (2012) The s2e platform: design, implementation, and applications. ACM Trans Comput Syst 30(1):2:1–2:49

Cong K, Lei L, Yang Z, Xie F (2015) Automatic fault injection for driver robustness testing. In: Proceedings of the 2015 international symposium on software testing and analysis, ISSTA 2015, New York. ACM, pp 361–372

Corteggiani N, Camurati G, Muench M, Poeplau S, Francillon A (2021) Soc security evaluation: reflections on methodology and tooling. IEEE Des Test 38(1):7–13

GNU (2022) Gnu coreutils – core utilities. https://www.gnu.org/s/coreutils

Godefroid P, Klarlund N, Sen K (2005) Dart: directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN conference on programming language design and implementation, PLDI'05, New York. ACM, pp 213–223

Godefroid P, Levin MY, Molnar DA (2012) SAGE: whitebox fuzzing for security testing. Commun ACM 55(3):40–44

Gu H, Chen M, Wei T, Lei L, Xie F (2018) Specification-driven automated conformance checking for virtual prototype and post-silicon designs. In: Proceedings of the 55th annual design automation conference, DAC 2018, San Francisco

Horn A, Tautschnig M, Val C, Liang L, Melham T, Grundy J, Kroening D (2013) Formal co-validation of low-level hardware/software interfaces. In: Proceedings of FMCAD

Intel (2009) Pci/pci-x family of gigabit ethernet controllers software developer's manual. https://www.intel.com/content/dam/doc/manual/pci-pci-x-family-gbe-controllers-software-dev-manual.pdf

Jakobs M, Pauck F, Platzner M, Wehrheim H, Wiersema T (2021) Software/hardware co-verification for custom instruction set processors. IEEE Access 9:160559–160579

Kannavara R, Havlicek CJ, Chen B, Tuttle MR, Cong K, Ray S, Xie F (2015) Challenges and opportunities with concolic testing. In: 2015 national aerospace and electronics conference (NAECON), pp 374–378

Kasikci B, Zamfir C, Candea G (2015) Automated classification of data races under both strong and weak memory models. ACM Trans Program Lang Syst 37(3):8:1–8:44

King JC (1976) Symbolic execution and program testing. Commun ACM 19(7):385–394

Kurshan RP, Levin V, Minea M, Peled D, Yenigün H (2002) Combining software and hardware verification techniques. Formal Methods Syst Des (FMSD) 21(3):251–280

Kuznetsov V, Kinder J, Bucur S, Candea G (2012) Efficient state merging in symbolic execution. In: Proceedings of the 33rd ACM SIGPLAN conference on programming language design and implementation, PLDI'12, New York. ACM, pp 193–204

Lattner C, Adve V (2004) LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the international symposium on code generation and optimization: feedback-directed and runtime optimization, CGO'04, Washington, DC. IEEE Computer Society, p 75

Lei L, Cong K, Yang Z, Chen B, Xie F (2019) Hardware/software co-monitoring. In: CoRR, arXiv:1905.03915 [cs.SE]

Li Z, Chen B, Feng W, Xie F (2021) Concolic execution of nmap scripts for honeyfarm generation. In: Jaeger T, Qian Z (eds) MTD@CCS 2021: proceedings of the 8th ACM workshop on moving target defense, virtual event, Republic of Korea, 15 Nov 2021. ACM, pp 33–42

Stoenescu T, Stefanescu A, Predut S, Ipate F (2016) RIVER: a binary analysis framework using symbolic execution and reversible x86 instructions. In: Fitzgerald JS, Heitmeyer CL, Gnesi S, Philippou A (eds) FM 2016: formal methods – 21st international symposium, Limassol, 9–11 Nov 2016, Proceedings. Volume 9995 of Lecture notes in computer science, pp 779–785

Luk CK, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K (2005) Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN conference on programming language design and implementation, PLDI'05, New York. ACM, pp 190–200

Lyu Y, Mishra P (2021) Scalable concolic testing of RTL models. IEEE Trans Comput 70(7):979–991

Marinescu PD, Cadar C (2012) Make test-zesti: a symbolic execution solution for improving regression testing. In: Proceedings of the 34th international conference on software engineering, ICSE'12, Piscataway. IEEE Press, pp 716–726

Mukherjee R, Purandare M, Polig R, Kroening D (2017) Formal techniques for effective co-verification of hardware/software co-designs. In: Proceedings of the 54th annual design automation conference, DAC 2017, Austin

Nethercote N, Seward J (2007) Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of the 28th ACM SIGPLAN conference on programming language design and implementation, PLDI'07, New York. ACM, pp 89–100

Palikareva H, Cadar C (2013) Multi-solver support in symbolic execution. In: Proceedings of the 25th international conference on computer aided verification, CAV'13. Springer, Berlin/Heidelberg, pp 53–68

Palikareva H, Kuchta T, Cadar C (2016) Shadow of a doubt: testing for divergences between software versions. In: Proceedings of the 38th international conference on software engineering, ICSE'16, New York. ACM, pp 1181–1192

Ramos DA, Engler D (2015) Under-constrained symbolic execution: correctness checking for real code. In: Proceedings of the 24th USENIX conference on security symposium, SEC'15, Berkeley. USENIX Association, pp 49–64

Redini N, Machiry A, Das D, Fratantonio Y, Bianchi A, Gustafson E, Shoshitaishvili Y, Kruegel C, Vigna G (2017) Bootstomp: on the security of bootloaders in mobile devices. In: 26th USENIX security symposium (USENIX security 17), Vancouver. USENIX Association, pp 781–798

Renzelmann MJ, Kadav A, Swift MM (2012) Symdrive: testing drivers without devices. In: Proceedings of the 10th USENIX conference on operating systems design and implementation, OSDI'12, Berkeley. USENIX Association, pp 279–292

Sen K, Marinov D, Agha G (2005) CUTE: a concolic unit testing engine for C. In: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on foundations of software engineering, ESEC/FSE-13, New York. ACM, pp 263–272

Shoshitaishvili Y, Wang R, Salls C, Stephens N, Polino M, Dutcher A, Grosen J, Feng S, Hauser C, Krügel C, Vigna G (2016) SOK: (state of) the art of war: offensive techniques in binary analysis. In: IEEE symposium on security and privacy, SP'16. IEEE Computer Society, pp 138–157

Song D, Brumley D, Yin H, Caballero J, Jager I, Kang MG, Liang Z, Newsome J, Poosankam P, Saxena P (2008) Bitblaze: a new approach to computer security via binary analysis. In: Proceedings of the 4th international conference on information systems security, ICISS'08. Springer, Berlin/Heidelberg, pp 1–25

Stephens N, Grosen J, Salls C, Dutcher A, Wang R, Corbetta J, Shoshitaishvili Y, Kruegel C, Vigna G (2016) Driller: augmenting fuzzing through selective symbolic execution. In: Proceedings of the network and distributed system security symposium, NDSS'16. The Internet Society

The Guardian (2017) IT meltdown has cost British Airways $80m so far, says Willie Walsh. https://www.theguardian.com/business/2017/jun/15/it-meltdown-cost-british-airlines-80m-so-far-willie-walsh-iag

The New York Times (2018) Facebook security breach exposes accounts of 50 million users. https://www.nytimes.com/2018/09/28/technology/facebook-hack-data-breach.html

Tianocore (2022) http://www.tianocore.org/

Tianocore (2022) EDK II. https://github.com/tianocore/edk2

Torvalds L (2005) Initial commit of linux kernel's git repository. https://git.io/fjGug

Wong E, Zhang L, Wang S, Liu T, Tan L (2015) Dase: document-assisted symbolic execution for improving automated software testing. In: Proceedings of the 37th international conference on software engineering – Volume 1, ICSE'15, Piscataway. IEEE Press, pp 620–631

Zheng H, Li D, Liang B, Zeng X, Zheng W, Deng Y, Lam W, Yang W, Xie T (2017) Automated test input generation for android: towards getting there in an industrial case. In: Proceedings of the 39th international conference on software engineering: software engineering in practice track, ICSE-SEIP'17, Piscataway. IEEE Press, pp 253–262

Information Flow Verification

39

Cynthia Sturton and Ryan Kastner

Contents

Introduction 1390
Information Flow 1391
  Information Flow Model 1391
  Specifying Information Flow Properties 1395
Information Flow Analysis 1396
  Trace Properties and Hyperproperties 1398
  Verifying Hyperproperties 1399
Verification Tools 1403
  Simulation-Based Verification 1403
  Formal Verification Methods 1404
Case Studies 1404
  Cache Timing Side Channels 1405
  Memory Access Control 1408
Conclusion 1410
References 1410

Abstract

Information flow tracking (IFT) models the movement of data which enables verification of security properties related to integrity and confidentiality. This chapter introduces the basics of hardware information flow analysis and illustrates its use for hardware security verification. The chapter starts by describing information flow models and properties. Then it highlights how information flow

C. Sturton (✉)
University of North Carolina at Chapel Hill, Chapel Hill, NC, USA
e-mail: csturton@cs.unc.edu

R. Kastner
University of California San Diego, La Jolla, CA, USA
e-mail: kastner@ucsd.edu

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_42

analysis is critical to hardware security verification. After that, there is a discussion regarding the differences between trace properties, which are commonly used in hardware functional verification, and information flow properties. This is followed by a description of different information flow verification techniques. The chapter concludes with two case studies demonstrating security verification using information flow properties.

Keywords

Information flow model · Hardware security properties · Hardware design verification · Cache timing side channels · Access control properties

Introduction

Security verification plays a crucial role in the development of modern hardware. Hardware vulnerabilities are difficult and expensive to fix if they are not caught early. Additionally, many security features are implemented in hardware, e.g., encryption, key management, authentication, and other hardware root of trust operations. Thus, semiconductor manufacturers and system integrators are increasing the amount of time and effort spent on hardware security verification.

Information flow tracking (IFT) is a fundamental technique that models how data propagates throughout a system. IFT verification aims to determine that only authorized flows of information are possible. IFT has been used to verify security of the cloud (Bacon et al. 2014), operating systems (Efstathopoulos et al. 2005; Krohn et al. 2007; Zeldovich et al. 2011), programming languages (Sabelfeld and Myers 2003), and hardware (Hu et al. 2021).

Hardware IFT allows verification of a wide range of security properties related to confidentiality, integrity, timing, availability, safety, hardware Trojans, and speculative execution (Hu et al. 2021). The key idea behind hardware IFT is that registers, memory locations, and other hardware state are given a security label in addition to their functional value. The IFT model gives rules on how to update these labels as the hardware executes. And the IFT verification tools provide analysis techniques to understand if, how, and when information flows occur through the hardware. A crucial step in the hardware security verification process is formally describing the security properties, i.e., defining the threat model. Property-driven hardware security (Hu et al. 2016) requires the verification engineer to specify the important storage locations or assets. The IFT tools verify how the information contained in an asset moves throughout the hardware.

Consider a common hardware security verification example – understanding how a confidential asset, e.g., a secret key, can move throughout the hardware. In this case, the key register would be tagged and the IFT tools used to understand how, when, and where that information flows. Typically the verification engineer will specify where that information should not flow by providing a security boundary or a set of disallowed sink registers or memory locations. For example, the key

information should not leak into user memory space. The IFT property may also involve conditions, e.g., the key cannot flow to the JTAG port except during debug mode. Properties related to integrity and timing can be crafted in a similar manner as described later in the chapter.

IFT verification involves understanding when, where, how, and why information flows occur. IFT verification techniques range from formal methods to simulation, emulation, and dynamic monitoring. Formal analysis provides guarantees on correctness and complete coverage but typically fails to scale past the level of IP cores. Simulation allows for larger analysis across a system on chip. Emulation allows for the even more complex analysis involving software and OS interactions. Dynamic monitoring performs real-time flow tracking.

This chapter serves to act as an introduction to the use of information flow tracking for hardware security verification. The aim is to provide the necessary background on information flow tracking models, properties, analysis, and verification tools and then demonstrate how information flow tracking can be used in two case studies related to cache timing leakage and on-chip access control.

Information Flow

Information flow tracking (IFT) models how information propagates through a computing system. More specifically, IFT provides the ability to label specific information in a system and understand how that information affects or moves throughout other parts of the system. The labels provide important metadata about the system state that can be used to understand where information can leak (confidentiality), how data has been modified (integrity), and the ability to affect timing behaviors (availability). Thus, IFT is a fundamental security verification technique as it allows one to reason about properties related to confidentiality, integrity, and availability.

Information Flow Model

Dorothy E. Denning pioneered the idea of information flow tracking (Denning 1976) defining an information flow model $\mathcal{IFT} = \langle \mathcal{N}, \mathcal{P}, \mathcal{SC}, \oplus, \rightarrow \rangle$ as a 5-tuple consisting of a set of storage objects $\mathcal{N}$, a set of processes $\mathcal{P}$, a set of security classes $\mathcal{SC}$, a class combining operator $\oplus$, and flow relation $\rightarrow$. IFT models are used in a variety of different abstraction levels including for hardware security verification (Hu et al. 2021). This article focuses on IFT in the context of computer architecture and therefore describes Denning's IFT model in that context. The *storage elements* consist of registers, memories, and other important architectural state. The *processes* are computations that the architecture performs, e.g., instructions, hardware accelerated functions, and interrupts. The *security classes* define how the storage objects can be labeled. When coupled with the *class combining operator*, security classes define how information flows as computation

occurs. The *flow relation* defines the allowable flows between every pair of security classes. For example, one might first label registers that hold cryptographic key material as "secret" and label ports as "public" and then further specify that information should never flow from "secret" to "public" elements. In this way, one can specify the flows that should and should not occur.

The security classes $\mathcal{SC}$ and their allowable flows $\rightarrow$ are key to using information flow tracking to verify security properties. Assume that there are two security classes: high and low ($\mathcal{SC} = \{H, L\}$) and a flow relation $L \rightarrow H$. The relation indicates that it is allowable for information labeled as L to flow to storage objects labeled as H. However, the opposite is not true: $H \nrightarrow L$; high information should never flow to storage objects with an L label. This simple lattice is shown in Fig. 1a.

While simple, this two-label lattice is quite powerful. Viewing the lattice in light of integrity (Fig. 1b), the H label would be considered *untrusted*, and the L label is *trusted*. In this case, one wants to ensure that *untrusted* information can never affect a trusted storage object, i.e., *untrusted* $\nrightarrow$ *trusted*, which would violate the integrity of the system. One can also view this same lattice through the lens of confidentiality (Fig. 1c) where *secret* information should never be leaked to an unclassified or openly viewable storage object.

Lattices can also be more complex. Figure 1d shows a lattice with eight different security classes: $\{A, B, C\}$, $\{A, B\}$, $\{A, C\}$, $\{B, C\}$, $\{A\}$, $\{B\}$, $\{C\}$, and $\emptyset$. The flow relation defines how information from three different categories, A, B, and C, should be allowed to move throughout the system. A label of $\{A, B, C\}$ indicates that this storage object has information from all three categories, the label $\{A\}$ indicates the information only comes from A, the label $\{B, C\}$ denotes the object has information from both B and C, and so on. The flow relation depicted in this lattice says that information from security class $\{A\}$, for example, is allowed to flow to security class $\{A, B\}$ and transitively to $\{A, B, C\}$ but should never flow to security class $\{B, C\}$. Lattices can be arbitrarily complex, although in practice the vast majority of IFT tools defaults to a two-label lattice as shown in Fig. 1a, b, and c.

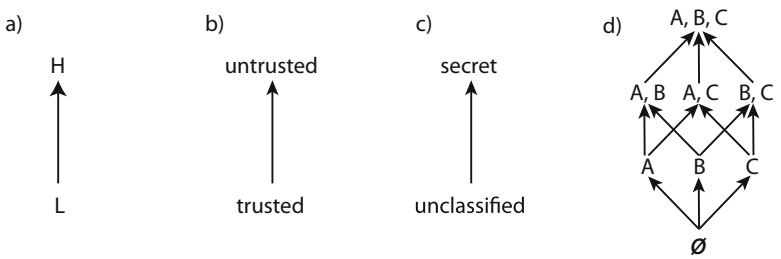


Fig. 1 A security lattice defines the flow relationships $\rightarrow$ between the different labels of a security class $\mathcal{SC}$. Part (**a**) defines a simple two-label $\{H, L\}$ security class with allowable flow from low L to high H. This same lattice can be used to define integrity (part **b**) and confidentiality (part **c**) properties. More complex lattices are possible, e.g., part (**d**) shows a more complicated lattice that uses 8 different labels to indicate the mixing of information between three entities A, B, and C. Most hardware security IFT tools use a simple two-label lattice like those in parts (**a**), (**b**), and (**c**)

A key idea of information flow tracking is that storage objects have a *security class label* in addition to their *functional value*. The label acts as an additional piece of metadata that IFT verification tools use to determine properties related to confidentiality, integrity, and availability.

IFT tools aim to provide some notion of tracking *noninterference* – an information flow model with a strict flow relation ($\rightarrow$) proposed by Goguen and Meseguer (1982). In the noninterference model of information flow, any changes in high inputs shall never be reflected in the low outputs. That is, low objects can learn nothing about high information. Another way to think about this is that a noninterfering computer system should produce the same low outputs for any set of low inputs regardless of the functional values of the high inputs, i.e., it is impossible to learn any information about the high values by controlling the low inputs and viewing the low outputs. Equivalently, the computer system, projected on to a "low view," responds exactly the same to any input sequence of low values regardless of the high values.

IFT determines how the labels propagate throughout the system by analyzing the system behavior and updating the labels corresponding to the storage objects. The IFT tool is given an initial labeling for the storage objects. Setting these initial labels determines which objects are considered high and which are considered low and therefore where information should and should not be allowed to flow. The rules for determining how an object's security class is updated are defined by the class combining operator $\oplus$.

IFT tools implement the class combining operators $\oplus$ to track information flows in different ways. The simplest and most conservative approach only considers the labels and marks the output of any process P as H when at least one of its inputs is H. In other words, the output of the process is labeled L only when all of the inputs to the process have an L label. In this case, the class combining operator is an OR gate. (This logic assumes that the label $H = 1$ and $L = 0$.) This is a safe approach but can lead to false positives where data is labeled as H when it should be L, i.e., the IFT tool states there is a flow when one does not exist.

The simple OR class combining operator is often too conservative and imprecise. It can quickly lead to many storage objects being marked as H even if they have no H information. To better understand the source of this imprecision, consider a process that implements a binary two-input Boolean AND operation as shown in Fig. 2. The inputs and outputs have their typical functional values (0 or 1). Additionally, each input/output has an associated label (H or L). The class combining operator is responsible for generating the output label. Part (a) shows the case when both inputs have an L label. The output label will be L regardless of the input's functional value. Part (b) illustrates a similar situation when the inputs all have an H label; the output will have an H label. More generally, if all inputs are H (L), the outputs will be H (L), respectively. The operator becomes more complex when one of the inputs is marked as H and the other input is L. Part (c) shows a case when one of the inputs is labeled L and has a functional value of 1. The other input has an H label. The results of the AND operation will be equal to the value of H's functional value. Thus, there is information about H in the output – changing

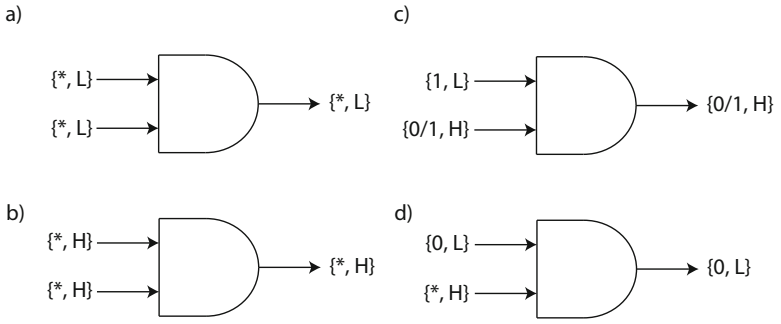


Fig. 2 A simple process implementing a binary two-input Boolean AND operation. The two inputs have the functional values (0/1) and the corresponding security class labels (H/L). The output label is determined by the class combining operator. Part (**a**) shows an example when both inputs have an L label. The output should be labeled at L regardless of the functional values of the inputs (denoted as *). Part (**b**) is similar where both inputs are H and the output label should be marked as H. Parts (**c**) and (**d**) show more complex examples when the input labels are mixed. Here the output labels depend on the functional values of the inputs. Part (**d**) shows the specific scenario when an output can be labeled L even if one of the inputs has an H label

H will result in a direct interference of the output of the AND operation. Thus, the output should be labeled H. Part (d) changes the functional value of the L input to 0. In this scenario, the output is always 0 regardless of the functional value of the H input. That is, no information related to the H input propagates to the output; thus the output can safely be labeled as L. Yet, the conservative approach would mark the output as H, i.e., stating that there is a flow from H to L when there is not. More complex combining operators consider the functional behavior of the process, the functional values of the inputs to that process, and their labels.

There are IFT models covering different types of flows – explicit and implicit (Denning and Denning 1977), timing (Oberg et al. 2014), and power (Nahiyan et al. 2020). This article focuses on explicit, implicit, and timing flows, which relate to the security properties that are more commonly verified in practice. The models differ on their security classes, class combining operator, and flow relations. Most of the variation is due to the class combining operator, e.g., determining explicit flows is generally much easier than determining implicit flows, which is generally easier than modeling timing flows. The basic ideas behind these different flows are described in the following. A hardware security verification engineer does not necessarily need to comprehend all the details how these flows are modeled – that is the job of the IFT tool – but they do require a basic understanding about the types of flows as those are important when specifying the information flow properties.

Figure 3 shows a simple example that illustrates the difference between *explicit flows* and *implicit flows*. The figure depicts a multiplexor as a process that has three inputs A, B, and S and one output O. Each input and output are associated with a storage object, e.g., a register/flip-flop. Explicit flows occur between the inputs

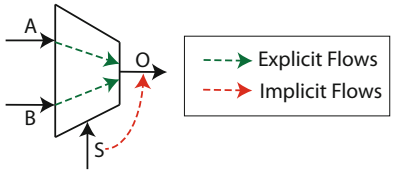


Fig. 3 A multiplexor exhibits explicit information flows between inputs A and B and the output O. There is an implicit flow between the selector input S and the output O

A and B and the output O. In this case, the values from A or B are directly copied into O. This means that O contains exact information about A and B, and thus O should take on the label of the input that is copied. The input S is the select bit which determines which of A or B to copy to the output O. There is an implicit flow between S and O due to the fact that an attacker would be able to determine information about the functional value of S by observing the functional values of O. Implicit information flows are more subtle but are still capable of being exploited.

A *timing flow* is a scenario where information is transferred based upon the time that a process take to compute an output. A common timing flow occurs with caches. An attacker will request some data from the cache. The data is returned quickly if it is stored in the cache. When the data is not in the cache, it takes a longer amount of time to fetch it from memory. The actual value of the data in both cases is the same, but the time at which it is delivered is different. Thus, if that presence/absence of that data in the cache was affected by another H process, e.g., the H process loads data that evicts some other data, then the attacker can ascertain H information. This is the core idea between Spectre (Kocher et al. 2019), Meltdown (Lipp et al. 2018), Foreshadow (Van Bulck et al. 2018) and other attacks that use the timing of cache operations to extract secret information.

Specifying Information Flow Properties

Information flow properties are specified by setting storage object labels and determining how those labels can and cannot flow throughout the system. Hardware security verification starts by developing a threat model and determining the security assets to protect (Aftabjahani et al. 2021). *Assets* are important system information whose behaviors should be monitored. A cryptographic key is a prime example of a security asset; a security verification engineer would want to understand how, when, and where information related to the key can move throughout the system. This is an example of a confidentiality property. Other examples of assets are control registers. Control registers often dictate important security scenarios. For example, a control register would be set in order to move the system into secure operating mode. Another control register would be set to indicate debug mode. Understanding who can set these control registers and the conditions under which they can be changed is important for the secure operation of the system. In this integrity scenario, one

would want to understand when the trusted registers could be influenced by some untrusted storage object.

To better understand how to use IFT for hardware security verification, consider the control/status register (CSR) associated with setting secure operating mode (e.g., the Secure Configuration Register from TrustZone). This register stores important information related to operating in a secure mode and is used to move into and out of secure operating mode. Thus, it is important to protect the integrity of this register. In this scenario one would mark common registers and memory space corresponding to the nonsecure world as untrusted and the CSR as trusted. One would want to determine if it is possible for the CSR label to ever become untrusted and, if it is, to understand the scenarios in which this can occur. IFT verification tools provide this ability.

IFT can also be used to understand security properties related to confidentiality. Consider the case where one aims to understand the confidentiality of a cryptographic key. IFT enables reasoning about how the value of a cryptographic key can flow throughout a computer system. In this scenario, assume the key is stored in a memory-mapped register associated with the custom IP core that performs the cryptographic operations. The system would be initially labeled in the following manner: the register that holds the key is labeled *secret*, and everything else in the system is labeled *unclassified*. IFT tools can help determine which parts of the system can learn information about the key. For example, can information related to the key ever flow to the cache? IFT verification would taint the key and attempt to ascertain the conditions under which a storage object related to the cache becomes labeled as *secret*, i.e., information related to the key has been leaked into the cache. Additionally, verification engineers often wish to make strict "no flow" conditions. For example, no information related to the key should ever leak into user space. Here one would taint the key and use IFT to see if the physical memory locations corresponding to the user space can ever be marked as *secret*. If they are, then there is some information about the key in those tainted memory locations.

More generally speaking, labels are assigned depending on the security properties under verification. The labels can be broadly interpreted to define properties related to confidentiality, integrity, and availability. At their most abstract, labels are defined by the security classes. The rules for calculating the labels are defined by the class combining operator. When combined with the flow relations, these define the types of security properties that IFT can verify.

Information Flow Analysis

Analyzing how information flows through a design is important for security; however, classical analysis and verification techniques cannot be used to assess information flow properties. To see why this is, consider the simple design of an AES block cipher shown in Fig. 4a. The AES module takes as input a cryptographic key (`key`) and the data to be encrypted (`data`) and outputs the resulting ciphertext (`ciphr`). A ready signal (`rdy`) indicates when the ciphertext is valid and can be

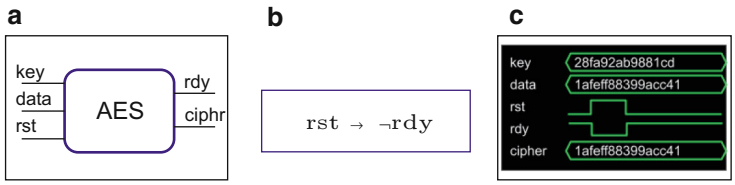


Fig. 4 A trace of execution can be used to confirm a simple assertion. (**a**) A simple AES module with inputs `key`, `data`, and `rst` and output `rdy`. (**b**) An assertion to capture correct reset behavior. (**c**) A trace of execution demonstrate the desired behavior

read. The ciphertext output is never valid while the module is undergoing a reset cycle, and the ready signal should reflect this. To test that the behavior of the ready signal is correct in this regard, one might write the simple assertion, $\texttt{rst} \rightarrow \neg\texttt{rdy}$ (Fig. 4b). Either traces of execution or a formal model of the design can then be studied to see whether it is indeed the case that the ready signal is never high during a reset cycle (Fig. 4c). Any trace-based verification technique can be used to either find violations of this property or determine that the behavior of `rdy` is likely correct because no violations are found (Because exhaustive state coverage is not feasible, a trace-based technique will not *prove* correctness with respect to a property for any but the simplest of designs. The most such a technique can verify is that no violations were found.). Alternatively, a formal verification technique such as model checking can be used to either find violations of the property or determine that no violations can occur within the first N number of clock cycles of execution (The bound N is an adjustable parameter to the verification tool.).

Another desirable property of the module is that information about the key should not flow to the ready signal. It should be impossible to recover even one bit of the `key` signal by studying the on-off behavior of the `rdy` signal. A variation in time to compute can reveal information about the value of `key` when that variation depends on the value of `key`. The behavior of `rdy` will reflect any such variation and therefore will reveal information about the key. When that happens, the `rdy` signal is said to *leak* information about `key`.

However, this property cannot be stated as a simple assertion in propositional logic. Let's say one wants to ensure that the value of the 0th bit of the `key` signal can never flow to the `rdy` signal. Two naively written properties might be $\texttt{key}_0 \rightarrow \texttt{rdy}$ or $\texttt{key}_0 \rightarrow \neg\texttt{rdy}$. However, neither one expresses the desired property, and in fact, both reflect acceptable behaviors of the design. It might be tempting to think that adding temporal logic to the property will solve the problem. For example, a property along the lines of $\texttt{key}_0 \rightarrow X(\texttt{rdy})$, which says that if the 0th bit of `key` is set then in the next (X) clock cycle `rdy` must also be set, seems to solve one problem with the naive properties, which is that information will surely take time to flow through a design. But, it does not solve the deeper problem, which is that there is no combination of values of $\texttt{key}_0$ and `rdy` that is illegal. Rather, it is that the value of `rdy` in a particular clock cycle should be the same whether `key` is set or unset; the value of `rdy` should not depend on the value of $\texttt{key}_0$. But, in order

to get at this property, it is not enough to reason about a single trace of execution. One needs to reason about traces in which $\texttt{key}_0$ is set *and* traces in which $\texttt{key}_0$ is unset. The property that is wanted is as follows: in all possible traces, whenever all of the inputs other than $\texttt{key}_0$ are fixed, the behavior of `rdy` will be fixed. In other words, if `data`, `rst`, and $\texttt{key}_1 \ldots \texttt{key}_{n-1}$ are fixed to particular values, the waveform of `rdy` will not vary, regardless of how $\texttt{key}_0$ varies. This defines noninterference between $\texttt{key}_0$ and `rdy` – nothing about the value of $\texttt{key}_0$ can be learned by observing `rdy`.

First-order logic (often abbreviated FOL) can be used to express the desired property that information should not flow from any of the bits of `key`. First-order logic is more expressive than propositional logic and allows one to state a notion of *for all* traces. First-order logic also allows one to introduce a notion of equality between two registers. The desired property, formally written, might look like this:

$$\forall\ \texttt{AES}_1, \texttt{AES}_2, \quad \texttt{rst}_1 = \texttt{rst}_2 \wedge \texttt{data}_1 = \texttt{data}_2 \rightarrow \texttt{rdy}_1 = \texttt{rdy}_2. \tag{1}$$

This property says that for any two traces produced by the AES module ($\forall\ \texttt{AES}_1, \texttt{AES}_2$), as long as `rst` has the same value in both traces and `data` has the same value in both traces, `rdy` will also have the same value in both traces (Note that one would probably like a stronger property saying that as long as `rst` carries the same value in both traces, regardless of both `data` and `key`, `rdy` will carry the same value in both traces.). The value of `key` does not affect the value of `rdy`. In other words, information does not flow from `key` to `rdy`. For simplicity, timing information has been elided; the true property would assert that the two `rdy` signals always have the same value at all points in the trace.

The key idea that makes the above property sound is that it reasons about *any* two possible traces. It is impossible to find two traces of the system in which `rst` and `data` are held fixed and the behavior of `rdy` varies.

Trace Properties and Hyperproperties

The example property about how information flows, written formally in Eq. (1), is fundamentally different than the example assertion about how signals behave, shown in Fig. 4b. The latter is an example of a *trace property*. It is a property that can be exhibited by a single trace of execution and can similarly be falsified by a single trace of execution. The types of properties that are expressible as SystemVerilog Assertions (SVA) are all trace properties. One usually thinks of these properties in terms of their logic formulation (e.g., $\texttt{rst} \rightarrow \neg\texttt{rdy}$), but another way to think about them is as a set of execution traces: for example, the set of all traces in which the statement $\texttt{rst} \rightarrow \neg\texttt{rdy}$ is valid. Using this definition in which a property is a set of traces, the property is true of a system if all of the traces the system could possibly produce are in the property's trace set.

The properties about how information flows, on the other hand, are not trace properties but rather *hyperproperties* (Clarkson and Schneider 2010). These properties are described by sets of traces, rather than by individual traces. Similarly, no single trace of execution can demonstrate a violation of a hyperproperty; only a set of two or more traces can do that. If a trace property is defined as a set of traces, then a hyperproperty is defined as a set of sets of traces, and every set of traces within the set represents a possible system satisfying the hyperproperty. Conversely, a hyperproperty is true of a system if all of the traces the system could possibly produce form one of the sets in the set of sets of traces that defines the hyperproperty. Hyperproperties can express notions of information flow. The AES hyperproperty is one example. Noninterference, described in section "Information Flow Model", is another example, and determinism, which says that only the defined inputs can affect the output (Roscoe 1995), is another. All of these information flow hyperproperties are important for security. Hyperproperties can also express notions of fairness, such as whether a coin flip has a non-biased outcome. However, this chapter focuses on properties related to information flow.

Going forward, the term *property* will be used in a generic sense to mean the behavior of the system. Where the distinction is important and not clear from the context, the terms *trace property* and *hyperproperty* will be used.

Verifying Hyperproperties

A strong security verification effort will require verifying information flow properties – hyperproperties – of a design. However, because a hyperproperty is neither satisfied nor falsified by any single trace of execution, the traditional verification efforts will not work. It is not possible to express hyperproperties in standard assertion specification languages such as SVA. But, even if it were possible – if the specification language was updated to include the *for all* quantifiers, for example – the traditional verification approaches would not be able to determine whether such a hyperproperty is valid of the design or not. Trace-based engines monitor the signals of the design as simulation progresses and look for any violation of the given assertion. The engine does not have knowledge of any prior or future simulation runs and therefore cannot reason about the comparative behavior of two or more traces of execution. Similarly, traditional model checking engines analyze the behavior of a single instance of the design and therefore cannot reason about the comparative behavior of two or more instances.

There are, however, new options for verifying information flow properties, and they can be categorized by whether they use a static or dynamic analysis technique. Under the static analysis category are cone-of-influence analysis and a more sophisticated model checking technique (section "Static Analysis"). Under the dynamic analysis category is information flow tracking (section "Dynamic Analysis"). Each of these are discussed in the following.

Static Analysis

In static analysis the RTL description of the design is itself analyzed. The analysis tool takes as input the RTL design, but rather than try to simulate the design, the analysis tool parses the design to answer a particular question.

Cone-of-influence analysis. The simplest form of static analysis that can be used to verify information flow properties is a cone-of-influence analysis. This analysis finds every signal in the design that can possibly affect the behavior of a signal or set of signals, and it is often used in the course of classical traditional verification to simplify the verification problem. COI analysis can be used to provide some insight into how information flows as follows. Suppose there is a design D with output signal `snk` (for "sink"), and the verification goal is to identify which information flows to this sink signal. First, a *COI set* is initialized to include the signal `snk`. The analysis then identifies every signal s which appears on the right-hand side of an assignment to `snk` in the RTL description of design D. Every newly identified s is added to the COI set. The analysis then repeats, for every s newly added to the COI set; every signal t which appears on the right-hand side of an assignment to s is added to the COI set. The analysis continues to repeat until a steady state is reached in which no new signals are added to the COI set. (As there are a finite number of signals in the design, the analysis is guaranteed to terminate.)

Every signal in the COI set has the potential to be a source of information flow to `snk`. This analysis is fast, requiring at most N iterations for a design with N unique signals, and the resulting COI set is complete: every signal which acts as a source of information flow to `snk` will be included in the COI set, or put another way, any signal that is not included in the COI set definitely does not influence the behavior of `snk`. However, the analysis is not sound: there may be signals included in the COI set which can never be a source of information flow to `snk`. Furthermore, the analysis provides a picture of how information flows in only broad strokes and details about the path that information takes through the design or the conditions under which flows occur are not possible with this analysis.

To better understand the limitations of COI analysis, consider Fig. 5. In Fig. 5a, an OR gate tied to 1 is connected to an AND gate. It is clear that while information from `B` and `C` both flow to, and affect the behavior of, output `O`, no information is flowing from `A` to `O` since the value of `B` is determined solely by the fixed input. However, a COI analysis will not capture that fact and will include `A` in the COI set of `O`. Figure 5b illustrates how details about which path information takes will be lost by COI analysis. In the top circuit of Fig. 5b, information flowing from `B` to `O` always passes through the XOR gate, while in the bottom circuit, information can flow directly from `B` to `O`. COI analysis cannot make that distinction, and the distinction is important for security. Suppose `A` is a one-bit secret key and `B` is a one-bit message that should be kept private. If the key is generated at random and without bias and is unknown to the observer at `O`, then the observer cannot learn any information about the message `B` even though there is information flowing from `B` to `O`: XORing the private message with the secret key obscures the information in

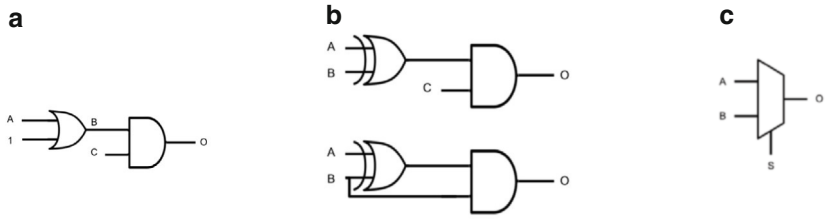


Fig. 5 A trace of execution can be used to confirm a simple assertion. (**a**) COI analysis would determine that information can flow from A, B, and C to O. However, there is no possible information flow from A to O. (**b**) In the top circuit, all information flowing from B to O always passes through the XOR gate, whereas in the bottom gate information can flow directly from B to O. COI analysis cannot differentiate between these two flows. (**c**) COI analysis would determine that information can flow from A, B, and S to O but would not determine the conditional nature of the flows from A or B to O

the message. However, in the bottom circuit, the observer at O will be able to learn information about B, for example, whenever the output at O is 1, the observer knows that the message is also 1. These two circuits have wildly different security postures, and in both cases, the analysis of information flow is important, but a COI analysis cannot provide the needed information. Finally, Fig. 5c illustrates a circuit with a conditional flow: depending on the value of S, information will flow either from A or B to O. (Information always flows from S to O.) However, the COI analysis will put both A and B in the COI set and has no way to make note of the conditional nature of the flow.

Model Checking Model checking is a formal verification technique that provides complete security validation of a design. With model checking, a design and a desired property are expressed as a logical formula which is checked for validity. As a simple example, consider Fig. 5a. The output of this circuit can be represented by the Boolean formula $(\mathtt{A} \vee 1) \wedge \mathtt{C}$. Let's suppose $\mathtt{C} \rightarrow \mathtt{O}$ is a desired property of the circuit. The model checking engine will search for a set of assignments to A and C that could violate this property. This is done by checking the satisfiability of the negation of the property: $\mathtt{isSAT}(\mathtt{C} \wedge \neg(\mathtt{A} \vee 1 \wedge \mathtt{C}))$. In this case, the SAT solver will be unable to find a satisfying solution to the formula, meaning the negation of the property cannot be satisfied and the desired property is true of the design. In reality the circuits to check are more complex, but the basic idea is the same: represent the circuit as a logical formula, and check whether the negation of the property can be satisfied given the constraints of the circuit. If so, the satisfying solution serves as a counterexample to the desired property – a set of inputs to the design that will violate the property. On the other hand, if no satisfying solution is found, the property is true of the design.

Practical circuits include sequential as well as combinational logic. Techniques for sequential model checking include bounded model checking and k-induction, in

which the design is unrolled so that a new logical formula representing the design is created for each clock cycle represented (Biere et al. 2003); IC3 or property-directed reachability, which does not require unrolling a design but instead reasons about the reachability of a "bad" state from a given state (Bradley 2011; Een et al. 2011); and engines based on binary decision diagrams (BDDs) which are used to efficiently represent sets of states and their transitions. The chapter on bit-level model checking describes the various techniques for sequential model checking in detail.

An out-of-the-box use of model checking in commercial verification engines cannot verify hyperproperties, which includes information flow properties. However, by carefully setting up the problem statement, model checking can be used to verify some types of hyperproperties called *k-safety* properties. Information flow properties are k-safety properties, and in particular, they are two-safety properties. The way to use model checking to verify two-safety properties is to use *self-composition*, a technique in which two identical instances of a design are combined in parallel to make one large design, which is then fed to the model checker (Terauchi and Aiken 2005). Going back to the AES example from earlier, two instances of the design, say $\texttt{AES}_1$ and $\texttt{AES}_2$, are combined to create AES. The model checker can then verify the property $\texttt{rst}_1 = \texttt{rst}_2 \wedge \texttt{data}_1 = \texttt{data}_2 \rightarrow \texttt{rdy}_1 = \texttt{rdy}_2$ of this combined design. Model checking is expensive, the size and complexity of the satisfiability queries grow quickly, and requiring two instances of a design doubles the starting complexity. For this reason, model checking information flow properties is limited to relatively small designs or to individual components of a design.

Dynamic Analysis

In dynamic analysis, it is the behavior of the design as it is simulated (or executed) that is analyzed. The analysis tool can be external to the design, in which case the tool can monitor only the input and output behaviors of the design. If, however, the behavior of internal signals needs to be analyzed, then the design itself must first be *instrumented* – modified in such a way that the logic needed for analysis is incorporated into the design itself.

Trace-based analysis methods do not apply to information flow properties, and dynamic analysis is a trace-based method: the behavior of a trace of execution is observed and analyzed by the tool. However, by using information flow tracking, dynamic analysis can be made applicable to the study of information flow properties (Tiwari et al. 2009). The design is instrumented to track how a particular input signal is affecting every other signal in the design. At the end of any single trace of execution, the added tracking logic has captured how information has flown from the input signal of interest.

To understand how this works, consider the simple AND gate in Fig. 6. In this example information flow tracking is used to expose how and when information from signal B can flow to output O. The original AND gate is on top, and the added tracking logic, in this case a second AND gate, is on the bottom. The new signals $\texttt{B}_\texttt{T}$ and $\texttt{O}_\texttt{T}$ track the information as it flows through the circuit. Going back to the Denning model of information flow, $\texttt{B}_\texttt{T}$, $\texttt{O}_\texttt{T}$, and the second AND gate are implementing the class combining operator $\oplus$ of the underlying model. Information

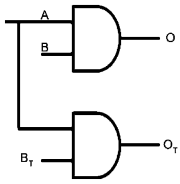


Fig. 6 The tracking logic for a simple AND gate

flows from `B` to `O` only when `A` is set; otherwise, the behavior of `O` is dominated by the unset `A` and unaffected by the behavior of `B`.

A design that has been instrumented with information flow tracking can then be run in simulation, while the tracking signals are monitored to look for any undesirable flows of information. Verification engines automate much of the instrumentation and monitoring, allowing engineers to focus solely on specifying the desired properties and providing comprehensive testbenches. Returning to our AES example, verifying that no information flows from `key` to `rd` using information flow tracking would require writing the property `key` $=/\Rightarrow$ `rdy`, where $=/\Rightarrow$ is the not-flow operator used by many information flow verification engines.

Information flow tracking does not suffer from the same issues of complexity that arise with model checking. On the other hand, it is a trace-based verification method and therefore cannot, in general, prove any property is true of a design but rather can only find instances when the property is violated.

The following sections dive deeper into the strengths and limitations of information flow tracking and how to use it in practice.

Verification Tools

Modern verification tools allow engineers to verify a variety of properties related to information flow through the design. Commercially available tools offer advanced simulation-based and formal method-based options for verifying information flow properties. In academia, new technologies are being developed, some of which will likely find their way into commercial tools down the line. This section presents and categorizes some of the current tools and discusses their strengths and weaknesses.

Simulation-Based Verification

The state of the art in simulation-based verification of information flow properties uses information flow tracking as described earlier. Source signals of interest are labeled, and the design is instrumented to track how data propagates from the source signals throughout the rest of the design. The designer provides an information flow property and testbench, the instrumented design is then simulated with the testbench providing input values, and the tool tracks whether the property is violated during the simulation run. An example property might state that information should flow from a particular source to a particular sink only when certain state conditions are

met. The designer must provide a testbench that sufficiently exercises the design to find possible property violations. Simulation-based verification tools cannot prove the correctness of a design; if no property violation is found, it is possible that the testbench was insufficiently complete. On the other hand, if a property violation is found, the root-cause analysis is simplified as the testbench provides the exact sequence of inputs that caused the property violation.

One commercial tool using information flow tracking is Radix-S from Tortuga Logic (Radix-S). The technology behind Tortuga Logic was first developed in academic research (for a survey, see Hu et al. 2021). Academic research has also demonstrated the use of information flow tracking to find timing channels in addition to data channels (Ardeshiricham et al. 2017).

Formal Verification Methods

The state of the art in formal verification of information flow properties uses a form of equivalence checking. The idea is similar in spirit to the use of self-composition with model checking described above and can be used to demonstrate determinism. The goal is to verify that a given destination signal is determined only by the known, allowed source signals. In other words, there is no additional, illegal flow of information from a given source signal to a given destination signal. In equivalence checking, two versions of a design are proven to exhibit the same behavior given the same environment and inputs. To verify determinism, two copies of a design are created, and in both, the known sources of information are constrained to be equal. Sequential equivalence checking is done to verify that the destination signal in both copies will be equal. If it is possible for the two copies to diverge, then there exists some additional path of information flow to the destination.

A benefit to formal verification methods is that the result is not dependent on testbench coverage. If a violation is not found, then a violation does not exist. However, in order to handle large-scale designs, engineers often have to introduce suitable abstractions, and finding the right abstraction can be challenging. In addition, if a violation is found, performing the root-cause analysis can be difficult.

Commercial tools that perform formal verification of information flow properties include JasperGold from Cadence (JasperGold), Questa Secure Check from Siemens (Questa Secure Check), and Formal Security Verification from Synopsys (VC Formal).

Case Studies

This section presents two case studies for performing security verification of a hardware architecture description. The first case study performs security verification of a cache specifically targeting timing side channels. The second case study verifies memory access control systems – an important aspect of modern secure computing systems. In each case, the threat model is described, the assets are defined, and example security properties are presented.

Cache Timing Side Channels

Caches are crucial for high-performance computer architectures and present in all but the simplest microprocessors. Caches take advantage of spatial and temporal dependencies in data access patterns. When the processor makes a memory transaction, it assumes that data and its neighboring values will likely be accessed again in the near future and thus keeps them in the faster, smaller cache memory. When a processor requests data that was already loaded into the cache, it is returned quickly (typically within a few cycles). When that data is not in the cache, the cache itself must request it from another slower but larger memory (which typically takes tens of cycles) during which time the cache stalls. The variation in the time of retrieving data results in a timing side channel (Percival 2005). This case study describes how to verify the existence of potential information leakage via a cache timing side channel. This leakage is powerful and a key element of Spectre (Kocher et al. 2019), Meltdown (Lipp et al. 2018), Foreshadow (Van Bulck et al. 2018), and other architectural security attacks.

Figure 7 shows a diagram of a cache. The cache sits between a processor (left interface) and another memory (right interface). The cache stores a number of cache `data` lines each with metadata that includes their valid `v` bit, its corresponding memory `tag`, and its associated processor ID `pid`. The PID is used to differentiate between users, secure/insecure mode, etc. The `Processor Transceive Logic` takes as input write data `wr`, a read/write address `addr`, write (`wr_req`) and read request (`rd_req`) signals, and a process ID `pid`. The outputs are the processor's requested read data `rd` and the `stall` signal indicating whether the cache is waiting for data. The `Memory Transcieve Logic` interfaces with another larger and slower memory, e.g., a lower-level cache or off-chip DRAM. This interface has an address `addr` and `rd` and `wr` busses that transfer cache lines between the cache and the memory. The terminology `processor.rd` and `memory.rd` are used when needed to disambiguate any unclear signal references.

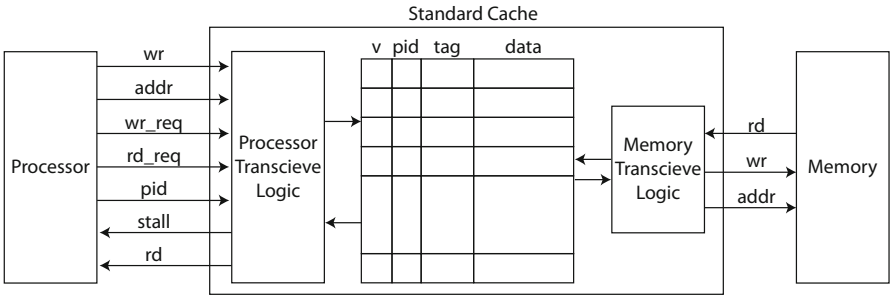


Fig. 7 The case study aims to understand potential threats related to a cache timing side channel attack. The security verification focuses whether information can leak from a sensitive process (PID i) to another untrusted process (PID j) via the timing behavior of a cache. IFT can determine such complex interactions

Assume that a threat model views a cache side channel as a security vulnerability. Thus, the security verification process aims to understand whether the cache is susceptible to an attack. The first responsibility is to identify the assets – what information needs protection, when is this information exposed to the cache, and where should it not flow? Specifying this in a manner that the IFT tool can understand involves interfacing with the security class labels. Digging further into the example will provide details on how and when to set and check the labels to verify a cache timing side channel.

Assume that the threat model requires that there is no timing side channel between PID i and PID j. Further, assume that PIDs are securely provided via the `pid` signal during processor read/write requests. One important cache side channel involves the leakage of information related to the memory accesses performed by some sensitive computation, in this case any computation performed by PID i. Thus, it is determined that `processor.addr` is an important asset that contains information that should not be leaked. `processor.addr` does not always carry information related to PID i; its label should only be marked high (H) when PID i is using the cache. Thus, the `addr_proc` label is set to H only if `pid == i`. The goal is to check when any sensitive address information about PID i flows from the cache to the processor while PID j is executing. `processor.rd` is one important signal where this information could flow. Thus, the analysis should assert that the `processor.rd` label is always L and report scenarios when it could become H as those are the conditions when information leakage about PID i's memory access pattern flows to PID j through a cache timing side channel. Finally, all storage objects (registers, Verilog variables, etc.) outside of the `addr_proc` signals should be initially marked as L.

IFT tools provide the capability to track different types of flows. In this case, the goal is to investigate timing flows, and thus, analysis requires an IFT tool that can track timing flows. Functional flows and timing flows differ in how the information is being transferred. *Functional flows* transfer information directly through the functional values of the storage object. Explicit flows are an example of a functional flow. *Timing flows* manifest themselves via the time at which the result is delivered.

In this cache timing side channel example, the goal is to know if the timing behavior of the cache provides any information to PID j about how PID i previously used the cache. A common timing flow manifests itself via the `processor.rd` signal, specifically the time at which valid data is presented on that signal. Note that there may be a timing flow without a functional flow since the value of the `processor.rd` will not vary (only the time at which that same value is delivered). These ideas are formalized by Oberg et al. (2014).

An IFT tool that tracks both functional flows and timing flows, e.g., Clepsydra (Ardeshiricham et al. 2017), will provide separate labels and check the status of the functional (f) and timing labels (t). The notation used here appends the labels to storage objects. For example, `processor.addr.f` and `processor.addr.t` indicate the functional flow label and timing flow label (respectively) for storage object `processor.addr`. Functional flows and timing flows are not independent. A functional flow is transferred to a timing flow in the case(s) when that functional

value is assigned conditionally. This causes a variation in the time at which that functional value is set, i.e., there is a timing flow.

Going back to the case study, the goal is to understand if the functional values `processor.addr` ever leak via a timing channel. Thus, the analysis requires setting `processor.addr.f = H` when PID i is using the cache and checking whether the H label can ever flow to `processor.rd.t` (the timing label of the read data) when `pid == j`.

Listing 1 provides an assertion-based IFT property for determining if a cache exhibits a timing side channel via its access patterns. The assertions work directly on the labels and use a labeling system similar to Clepsydra (Ardeshiricham et al. 2017) for verifying functional and timing flows. The default functional `f` and timing `t` labels are set to L. The `processor.addr.f` functional label is set to H when PID i is using the cache, indicating that this information will be tracked. The analysis must then determine if any information about how PID i used the cache is leaked to `processor.rd` via a timing channel. This is reflected in the `processor.rd.t` value. If it is H, that indicates that `processor.rd` contains some information about `processor.addr` via a timing channel.

Listing 1 Assertion-based property covering cache timing side channel from Fig. 7

```
assume (default f,t == L);
if (pid == i)
        assume (processor.addr.f == H);
if (pid == j)
        assert (processor.rd.t == L)
```

All IFT tools provides a manner for setting and checking the IFT labels. That may be done directly, as shown above, or it could be through some higher-level property specification. One common IFT property language feature is the "no flow" operator `=/=>` which indicates that information from a source storage object should not flow to a sink storage object, i.e., `source =/=> sink`. This is equivalent to setting the `source` label H and verifying that `sink` label stays L.

In addition to the `=/=>` operator, IFT tools often have a way to specify the conditions under which information should be tracked (i.e., when to set the source label) and conditions under which flows are allowable (i.e., when to check if the `sink` label is H). Listing 2 provides cache side channel property using the no-flow operator:

Listing 2 No-flow property covering cache timing side channel from Fig. 7

```
processor.addr.f when (pid == i) =/=> processor.
rd.t unless (pid == i)
```

This is equivalent to Listing 1 but provides a higher-level abstraction for flow modeling. The `when` keyword provides the time to mark `processor.addr` as H, and the `unless` keyword denotes that flow is allowable when `pid == i`, but at no other times.

Regardless of how the property is specified, a standard cache without any timing side channel mitigation should fail security verification related to a timing channel. That is, the verification process will find at least one scenario when information about PID i memory accesses are leaked to PID j via a timing channel on the `processor.rd` object. Without any mitigation in place, that scenario would occur when PID j accesses a cache line that was previously accessed by PID i.

Memory Access Control

Security-critical applications often have dynamic policies for securing memory transactions. Examples include the following: (1) a memory region is temporarily isolated when an untrusted application is executing, (2) kernel state is only accessible during debug mode, and (3) only the hardware root of trust can access and set security-critical control and status registers.

In order to operate in a secure manner, processors use some form of an *access control system* that enforces an *access control policy*. The access control policy defines how different components of a processor (CPU core, custom accelerators, hardware root of trust, peripherals, memory management units, etc.) can access each other. The access control policy changes over the lifetime of the system – moving from the manufacturer, to the system integrators, and finally to the end users. Policies often change at runtime, e.g., different access restrictions occur during boot mode, secure operating modes, reset, debug, and normal operating modes.

The access control system plays a crucial role in maintaining secure computing environments and must undergo a rigorous verification process to ensure that it is implemented correctly. Verification includes functional correctness. Additionally, and equally as important, the access control system must undergo a security verification that addresses potential security weaknesses and vulnerabilities. An exploit in the access control system endangers the confidentiality, integrity, and availability of the overall system.

Unfortunately, it is challenging to correctly implement access control systems. The MITRE Common Weakness Enumeration (CWE) database reports a substantial and growing number of hardware weaknesses (The Common Weakness Enumeration Official Webpage 2022). Restuccia et al. performed a systematic review and found 30 CWEs related to access control (Restuccia et al. 2021). These weaknesses range from ensuring that configuration registers are properly reset, that interrupts are correctly handled, and that memory regions are isolated. The following provides more detail on some of these properties.

In order to better understand how to perform security verification, consider the on-chip access control module depicted in Fig. 8. An access control wrapper is programmed by a trusted entity, e.g., a hardware root of trust, to restrict the memory accesses of a CPU core to other memory-addressable resources (different processors, custom accelerators, peripherals). The wrapper inspects the CPU core's read/write requests and only passes along requests that adhere to the access control manager configuration. A simple configuration would be to only allow address

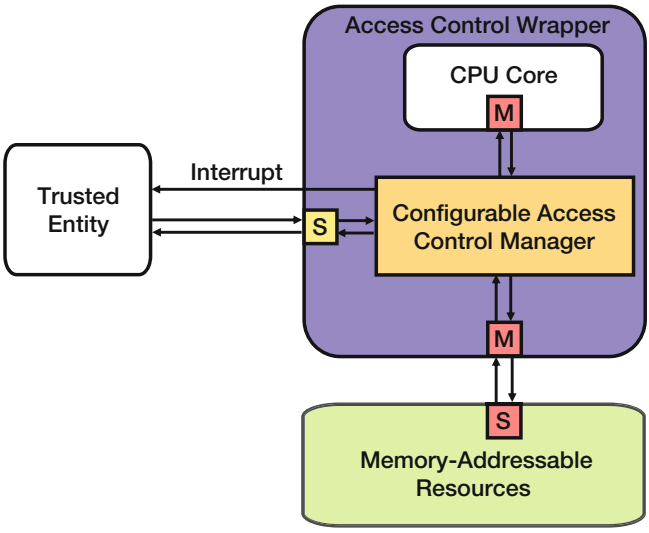


Fig. 8 The access control wrapper inspects memory requests from the CPU core and relays them to the memory interconnect only if they adhere to the policy set in the configurable access control manager, which is programmed by a trusted entity. Any illegal accesses are stopped at the source and the trusted entity is alerted via an interrupt

requests between a specified low address `BASE_LOW_ADDR` and high address `BASE_HIGH_ADDR`. If an illegal request is made, the access control wrapper takes an action, e.g., decoupling the IP core from the on-chip interconnect and sending an interrupt to the trusted entity. The access control wrapper uses standard AXI interfaces where a manager *M* initiates transactions and a subordinate *S* responds to the access requests.

Security-critical properties of the access control module include both trace and hyperproperties. An example trace property is shown in Listing 3, which states that the AXI write address channel `AW` is always disabled after an interrupt occurs. This property can be gleaned by studying individual traces of execution. In each trace, whenever the `interrupt` signal is set, the `AW_CH_EN` signal will be unset. Identifying the pattern immediately gives rise to the property. It is also the case that the property can be disproved by a single trace of execution – one where both `interrupt` and `AW_CH_EN` are set.

Listing 3 A trace property that states the AXI manager *M* address write channel is properly disabled during an interrupt

```
interrupt |-> !AW.CH.EN
```

An example information flow hyperproperty is shown in Listing 4, which states that any information coming via the CPU core's write port (`CPU.M.WDATA`) during

an illegal write request (Indicated here by the pseudo signal `illegal_request`) should not be reflected in the access control output write channel (`ACW.M.WDATA`).

Listing 4 An information flow property that states that information from the CPU core manager *M* interface should never flow outside of the access control wrapper when that information corresponds to an illegal request

```
CPU.M.WDATA when illegal_request =/=> ACW.M.WDATA
```

Researchers have defined over 300 properties related to basic security behaviors of the access control wrapper; those properties along with the hardware description of the access control wrapper are available in the Aker open-source repository (Aker Github Repository). These properties were inspired by MITRE Common Weakness Enumerations (CWEs) (The Common Weakness Enumeration Official Webpage 2022) specifically those related to hardware design and include a mix of trace properties and hyperproperties.

Property generation was far and away the most challenging, time-consuming, yet important part of the security validation process. Automating property generation is invaluable in making security validation faster and more comprehensive, and new research has begun to do just that (Deutschbein et al. 2021, 2022; Zhang et al. 2017). Potentially even more valuable is providing the engineer with insights into the design under validation, which could lead to the specification of additional properties and the discovery of new weaknesses and vulnerabilities.

Conclusion

This chapter demonstrates the benefits of a property-driven hardware security flow for verifying the security of hardware designs. Information flow tracking plays a crucial role in hardware security verification; information flow models and properties provide basic formalization of flow models and a language to specify security properties. IFT allows for the definition and verification of hyperproperties, which provide a way to reason about noninterference – an important aspect for verifying properties related to confidentiality and integrity. There are different tools and techniques for analyzing information flows each with their own strengths and weaknesses. The chapter concludes with two case studies demonstrating how to perform security verification for a cache timing side channel and on-chip access control.

References

Aftabjahani S, Kastner R, Tehranipoor M, Farahmandi F, Oberg J, Nordstrom A, Fern N, Althoff A (2021) Special session: Cad for hardware security-automation is key to adoption of solutions. In: 2021 IEEE 39th VLSI Test Symposium (VTS). IEEE, pp 1–10

Aker Github Repository. https://github.com/KastnerRG/AKER-Access-Control

Ardeshiricham A, Hu W, Kastner R (2017) Clepsydra: modeling timing flows in hardware designs. In: 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). IEEE, pp 147–154

Bacon J, Eyers D, Pasquier TF-M, Singh J, Papagiannis I, Pietzuch P (2014) Information flow control for secure cloud computing. IEEE Trans Netw Serv Manag 11(1):76–89

Biere A, Cimatti A, Clarke EM, Strichman O, Zhu Y (2003) Bounded model checking

Bradley A (2011) SAT-based model checking without unrolling

Clarkson MR, Schneider FB (2010) Hyperproperties. J Comput Secur 18(6):1157–1210. http://dl.acm.org/citation.cfm?id=1891823.1891830

Denning DE (1976) A lattice model of secure information flow

Denning DE, Denning PJ (1977) Certification of programs for secure information flow. Commun ACM 20(7):504–513

Deutschbein C, Meza A, Restuccia F, Kastner R, Sturton C (2021) Isadora: automated information flow property generation for hardware designs. In: Proceedings of the Workshop on Attacks and Solutions in Hardware Security (ASHES). ACM

Deutschbein C, Meza A, Restuccia F, Gregoire M, Kastner R, Sturton C (2022) Toward hardware security property generation at scale. IEEE Secur Privacy. Special issue: Formal Methods at Scale

Een N, Mishchenko A, Brayton R (2011) Efficient implementation of property directed reachability. In: 2011 Formal Methods in Computer-Aided Design (FMCAD), pp 125–134

Efstathopoulos P, Krohn M, VanDeBogart S, Frey C, Ziegler D, Kohler E, Mazieres D, Kaashoek F, Morris R (2005) Labels and event processes in the asbestos operating system. ACM SIGOPS Oper Syst Rev 39(5):17–30

Goguen JA, Meseguer J (1982) Security policies and security models. In: IEEE Symposium on Security & Privacy, pp 11–20

Hu W, Althoff A, Ardeshiricham A, Kastner R (2016) Towards property driven hardware security. In: 2016 17th International Workshop on Microprocessor and SOC Test and Verification (MTV). IEEE, pp 51–56

Hu W, Ardeshiricham A, Kastner R (2021) Hardware information flow tracking. ACM Comput Surv (CSUR) 54(4):1–39

JasperGold. Cadence. [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html

Kocher P, Horn J, Fogh A, Genkin D, Gruss D, Haas W, Hamburg M, Lipp M, Mangard S, Prescher T et al (2019) Spectre attacks: exploiting speculative execution. In: 2019 IEEE Symposium on Security and Privacy (SP). IEEE, pp 1–19

Krohn M, Yip A, Brodsky M, Cliffer N, Kaashoek MF, Kohler E, Morris R (2007) Information flow control for standard OS abstractions. ACM SIGOPS Oper Syst Rev 41(6): 321–334

Lipp M, Schwarz M, Gruss D, Prescher T, Haas W, Fogh A, Horn J, Mangard S, Kocher P, Genkin D et al (2018) Meltdown: reading kernel memory from user space. In: *27th* {USENIX} Security Symposium ({*USENIX*} *Security 18)*, pp 973–990

Nahiyan A, Park J, He M, Iskander Y, Farahmandi F, Forte D, Tehranipoor M (2020) Script: a cad framework for power side-channel vulnerability assessment using information flow tracking and pattern generation. ACM Trans Des Autom Electron Syst (TODAES) 25(3):1–27

Oberg J, Meiklejohn S, Sherwood T, Kastner R (2014) Leveraging gate-level properties to identify hardware timing channels. IEEE Trans Comput-Aided Des Integr Circuits Syst 33(9):1288–1301

Percival C (2005) Cache missing for fun and profit

Questa Secure Check. Siemens. [Online]. Available: https://eda.sw.siemens.com/en-US/ic/questa/formal-verification/secure-check/

Radix-S. Tortuga Logic. [Online]. Available: https://tortugalogic.com/

Restuccia F, Meza A, Kastner R (2021) Aker: a design and verification framework for safe and secure SOC access control. In: IEEE/ACM International Conference on Computer-Aided Design (ICCAD). IEEE

Roscoe A (1995) CSP and determinism in security modelling. In: Proceedings 1995 IEEE Symposium on Security and Privacy, pp 114–127

Sabelfeld A, Myers AC (2003) Language-based information-flow security. IEEE J Sel Areas Commun 21(1):5–19

Terauchi T, Aiken A (2005) Secure information flow as a safety problem. In: Proceedings International Static Analysis Symposium. Springer, pp 352–367

The Common Weakness Enumeration Official Webpage (2022). MITRE, https://cwe.mitre.org/

Tiwari M, Wassel HM, Mazloom B, Mysore S, Chong FT, Sherwood T (2009) Complete information flow tracking from the gates up. In: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, pp 109–120

Van Bulck J, Minkin M, Weisse O, Genkin D, Kasikci B, Piessens F, Silberstein M, Wenisch TF, Yarom Y, Strackx R (2018) Foreshadow: extracting the keys to the intel {SGX} kingdom with transient out-of-order execution. In: 27th {USENIX} Security Symposium ({USENIX} Security 18), pp 991–1008

VC Formal. Synopsys. [Online]. Available: https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html

Zeldovich N, Boyd-Wickizer S, Kohler E, Mazieres D (2011) Making information flow explicit in histar. Commun ACM 54(11):93–101

Zhang R, Stanley N, Griggs C, Chi A, Sturton C (2017) Identifying security critical properties for the dynamic verification of a processor. In: Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). ACM

Verification of Quantum Circuits

40

Robert Wille and Lukas Burgholzer

Contents

Introduction 1414
Background 1416
Quantum Computing 1416
Quantum Circuit Compilation 1418
Verification 1420
Classical Circuits 1420
Quantum Circuits 1421
Formal Verification 1423
Decision Diagrams 1424
General Approach 1425
Alternating Approach 1426
Designing a Strategy for Verifying Compilation Flow Results 1427
Simulative Verification 1430
Verification Schemes Based on Simulation 1431
Stimuli Generation Schemes 1432
Resulting Quantum Circuit Equivalence Checking Flow 1436
Conclusions 1438
References 1438

R. Wille (✉)
Chair for Design Automation, Technical University of Munich, Munich, Germany
Software Competence Center Hagenberg GmbH (SCCH), Hagenberg im Mühlkreis, Austria
e-mail: robert.wille@tum.de; robert.wille@jku.at

L. Burgholzer
Institute for Integrated Circuits, Johannes Kepler University Linz, Linz, Austria
e-mail: lukas.burgholzer@jku.at

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3_43

Abstract

We are at the dawn of a new "computing age" in which quantum computers will find their way into practical applications. Although quantum computers work differently than classical machines, the design flow for realizing applications is similar: first, the desired functionality/application is described on a high level. Then, it is compiled down to a description (usually called *quantum circuit*) that can be executed on an actual machine. During this process, lots of constraints have to be fulfilled, and optimizations are applied to reduce the circuit's size and, hence, improve the actual performance on the quantum computer – all of which are highly nontrivial steps. As in conventional design, sooner or later, it is essential to check whether the resulting realization is correct – motivating verification. This chapter reviews and provides a summary of work in this regard. Considering the challenges currently seen in the verification of (comparatively simpler) classical systems, this may provide the basis for preventing the emergence of a verification gap in quantum computing.

Keywords

Quantum computing · Verification · Equivalence checking · Decision diagrams

Introduction

In the 1970s, researchers started to utilize quantum mechanics to address questions in computer science and information theory – establishing new research directions such as quantum computing, quantum information, and quantum security (Nielsen and Chuang 2010). In all these fields, quantum bits (i.e., qubits) serve as elementary information unit, which – in contrast to classical bits – can not only be in one of its two orthogonal basis states (denoted $|0\rangle$ and $|1\rangle$ using Dirac notation) but also in a superposition of both (i.e., $\alpha_0 |0\rangle + \alpha_1 |1\rangle$, where the complex amplitudes $\alpha_0, \alpha_1 \in \mathbb{C}$ satisfy $\alpha_0\alpha_0^* + \alpha_1\alpha_1^* = 1$). This allows an n-qubit quantum system to represent a (linear) combination of all 2^n different n-bit values at once – exponentially more than classical n-bit systems which can only represent a single n-bit value at a time. Together with further quantum-physical phenomena such as entanglement (Nielsen and Chuang 2010), this allows for substantial improvements in information density as well as computational power and motivated the establishment of dedicated research areas in computer science and information theory investigating and exploiting this potential.

Applications for quantum computers are usually described in terms of high-level quantum algorithms provided in dedicated quantum programming languages such as *Quipper* (Green et al. 2013), *Q#* (Svore et al. 2018), and *OpenQASM* (Cross et al. 2021) (comparable to high-level programming languages such as C++, Java, Haskell, and assembly languages for the classical realm). In order to execute the algorithms, these descriptions have to be broken down into elementary operations

(e.g., a sequence of dedicated microwave pulses applied to the qubits) that can be executed on a real quantum computer (Amy et al. 2013; Barenco et al. 1995; Maslov 2016; Giles and Selinger 2013; Zulehner and Wille 2018) – usually called *quantum circuit*. Additionally, physical constraints of the target hardware have to be taken into account, e.g., not all qubits may directly interact with each other and different operations take different amounts of time (Zulehner et al. 2019d; Smith and Thornton 2019; Wille et al. 2019; Li et al. 2019; Matsuo et al. 2019; Murali et al. 2019; Siraichi et al. 2018; Amy and Gheorghiu 2019; Zulehner and Wille 2019a; Burgholzer et al. 2022a). Finally, several optimizations are applied to reduce the circuit's size and, hence, increase the expected fidelity when executing the circuit on the actual quantum computer (Itoko et al. 2020; Vidal and Dawson 2004; Nam et al. 2018; Hietala et al. 2019). These tasks are often referred to as *compilation* (since eventually an "assembly program" results), *synthesis* (since quantum circuits often serve as means of description), *decomposition* (since high-level operations are broken down to elementary operations), or *mapping* (since qubits of the algorithms are mapped to the physical qubits of the target architecture).

All these steps (in the following summarized as *compilation*) result in different representations of the considered functionality, which significantly differ in their basis operations and structure but are still supposed to be functionally equivalent. Consequently, checking whether the original functionality is indeed maintained throughout all these different abstractions becomes increasingly relevant in order to guarantee a consistent and error-free design flow. This is similar in the classical realm where, e.g., descriptions at the *electronic system level*, the *register transfer level*, and the *gate level* exist. Here, these descriptions are verified using design automation expertise leading to efficient methods for verification (more precisely, for *equivalence checking*) in order to guarantee correctness throughout the design. However, since quantum circuits additionally employ quantum-mechanical effects, such as superposition and entanglement, those methods cannot be used for verification in the quantum realm in an out-of-the-box fashion. Accordingly, how to do verification for quantum circuits has to be approached from a different perspective.

At a first glance, these quantum characteristics make the problem of verification harder – suddenly, circuits have to be supported which do not only rely on 0s and 1s but also on superposition or entanglement. And indeed, this task has been proven to be QMA-complete (The class QMA is the natural extension of the classical class NP to the quantum computing world (Bookatz 2013).) (Janzing et al. 2005). But, at the same time, the inherent reversibility of quantum computations offers potential not available in classical computing. More precisely:

- It allows for *formal* equivalence checking of two circuits by proving that the composition of one circuit with the inverse of the other implements the identity – a structure that can be represented very efficiently. If conducted in a clever fashion, this efficient representation can be maintained throughout the entire verification process. Eventually, *alternating* between applications of gates from either circuit allows to conduct verification in an efficient fashion (under the assumption that a suitable oracle can be derived).

- It enables even small differences in quantum circuits to frequently affect the entire functional representation. Hence, the simulation of both computations with a couple of arbitrary input states, i.e., considering only a small part of the whole functionality, provides an attractive alternative to formal verification as described above.

This chapter sheds light into the challenges of quantum circuit verification but also how characteristics such as those above can be utilized to tackle them. To this end, section "Background" provides the necessary background to keep this chapter self-contained. Then, section "Verification" formulates the problem of verifying quantum circuits. Afterward, sections "Formal Verification" and "Simulative Verification", respectively, describe the formal and simulation-based verification techniques for quantum circuits that make use of the characteristics mentioned above. Eventually, the composition of both techniques into the first advanced equivalence checking flow is described in section "Resulting Quantum Circuit Equivalence Checking Flow". Implementations of all the methods provided in this chapter are publicly available as open-source and can be accessed at github.com/cda-tum/qcec. By this, the chapter provides the basis for preventing the emergence of a verification gap in quantum computing as it is currently present for classical circuits.

Background

This section briefly reviews the concepts of quantum computing and quantum circuit compilation. For more detailed information, the interested reader is referred to the provided references.

Quantum Computing

In quantum computing (Nielsen and Chuang 2010), the main computational unit is the *qubit*. In contrast to classical bits, a single qubit q can be in an arbitrary superposition of the basis states $|0\rangle$ and $|1\rangle$, i.e.:

$$|q\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle$$

with $\alpha_0, \alpha_1 \in \mathbb{C}$ and $|\alpha_0|^2 + |\alpha_1|^2 = 1$. An n-qubit quantum system can be in an arbitrary superposition of the 2^n basis states:

$$|b_{n-1}\rangle \otimes \cdots \otimes |b_0\rangle = |b_{n-1} \ldots b_0\rangle = |\sum_{i=0}^{n-1} b_i 2^i\rangle$$

with $b_i \in \{0, 1\}$, i.e.:

$$|q\rangle_n = \sum_{i=0}^{2^n-1} \alpha_i \, |i\rangle \text{ with } \alpha_i \in \mathbb{C} \text{ and } \sum_{i=0}^{2^n-1} |\alpha_i|^2 = 1.$$

In the circuit model of quantum computation, qubits are represented by *wires* and are manipulated by quantum operations (*quantum gates*). Specifically, a quantum circuit G with m gates, operating on n qubits, is denoted by $G = g_0 \dots g_{m-1}$, where each g_i represents a quantum gate acting on (a subset of) n qubits. This is usually visualized through *quantum circuit diagrams*, where the qubit wires are drawn as horizontal lines, gates are drawn using a variety of symbols, and progression of time is assumed to happen from left to right.

Example 1. An example of a quantum circuit G with 16 gates acting on three qubits is shown in Fig. 1a. This sequence of operations describes a small instance of the famous *Grover* search algorithm (Grover 1996). The small boxes with identifiers correspond to operations applied to single qubits such as X gates (the quantum analogue to the *NOT* gate) and H(Hadamard) gates (which can be used to set a qubit into superposition). Moreover, there are *multiple-controlled X* operations, where an X operation is only applied to a *target* qubit (denoted by $\oplus$) if all of its *control* qubits

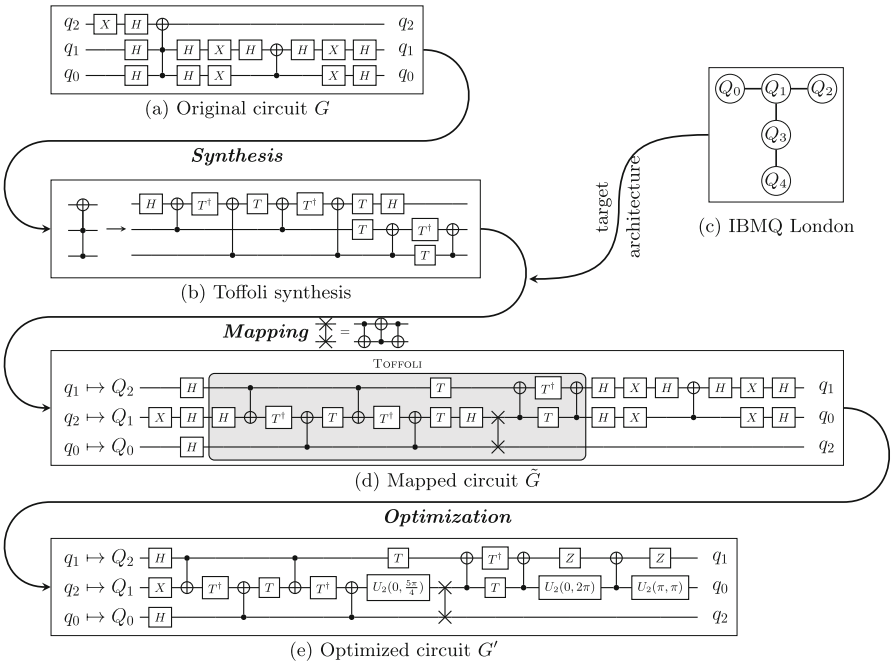


Fig. 1 Exemplary illustration of the quantum circuit compilation flow

(denoted by •) are in state $|1\rangle$. In case there is only one control qubit, such a gate is also called *CNOT* or controlled-*NOT*, while in case of two control qubits, it is also called a *Toffoli* gate.

Each gate g_i represents a corresponding unitary matrix U_i that is subsequently applied during the execution of a quantum circuit. Thus, the functionality of a given circuit $G = g_0 \dots g_{|G|-1}$ can be obtained as a unitary *system matrix* U itself by determining $U = U_{|G|-1} \cdots U_0$. Moreover, executing the quantum circuit for a given initial state $|\varphi\rangle$ (also called *classical simulation* when conducted on a conventional computer) leads to an evolution of the state $|\varphi\rangle$ according to $U_{|G|-1} \cdots U_0 |\varphi\rangle = U |\varphi\rangle = |\varphi'\rangle$ (If $|\varphi\rangle = |i\rangle$ for some $i \in \{0, \dots, 2^n - 1\}$, i.e., $|\varphi\rangle$ is a computational basis state, the simulation precisely calculates the ith column u_i of U, i.e., $U |i\rangle = |u_i\rangle$.).

Quantum Circuit Compilation

Initially, quantum algorithms are described in a way which is agnostic of the device they are planned to be executed on. However, physical devices today impose several constraints on the circuits to be executed. Thus, just as in classical computing, a conceptual algorithm needs to be *compiled* to the targeted architecture. Compilation of quantum circuits addresses three kinds of restrictions which limit the usability of a quantum computer:

1. **Limited gate-set:** Typically, only a small set of gates is natively supported by devices, e.g., consisting of arbitrary single-qubit gates and the two-qubit *CNOT* operation.
2. **Limited connectivity:** Devices frequently limit the pairs of qubits that operations may be applied to. This is usually described by a *coupling graph*, where the graph's nodes represent the qubits and an edge between two nodes indicates that a *CNOT* operation may be applied to those qubits.
3. **Short coherence times** and **limited fidelity:** A device's physical qubits are inherently affected by noise. Until a certain threshold concerning the number of available qubits is reached, error correction is not yet an option.

The first two, i.e., the limited gate-set and connectivity, constitute *hard* constraints – a computation not conforming to these restrictions may not be executed on the device. In contrast, the short coherence time and limited gate fidelity represent *soft* constraints – a quantum circuit may be executed on a device, but it is not guaranteed to produce meaningful results if the circuit, e.g., is too large for the state to stay coherent.

In order to tackle these limitations, first, the gates of the original quantum circuit are **synthesized** to the gate-set supported by the targeted device. Most importantly, since devices typically only support up to two-qubit gates, any gate acting on more than two qubits is broken down into "elementary" gates. This process may require the use of *ancillary* qubits for realizing the desired operation, e.g., for the

decomposition of multi-controlled gates – offering a trade-off between circuit size and number of required ancillary qubits (Amy et al. 2013; Barenco et al. 1995; Maslov 2016; Giles and Selinger 2013; Zulehner and Wille 2018).

Example 2. Consider again the circuit G from Example 1 as shown in Fig. 1a. If this circuit shall be executed on a system that only supports arbitrary single-qubit gates and *CNOT*s, the Toffoli gate (the two-controlled *NOT*) first has to be decomposed into this gate-set. One possible synthesized version is shown in Fig. 1b. It takes six *CNOT*s, nine single qubit gates, and no additional ancillaries to realize the desired gate.

Now, the circuit just contains elementary gates supported by the device, but it may not yet conform to the device's limited connectivity. Thus, the quantum circuit is **mapped** to the target architecture, i.e., a mapping between the circuit's *logical* and the device's *physical* qubits is established. Several heuristics for determining a suitable *initial mapping* exist – from a trivial one-to-one mapping ($q_i \mapsto Q_i$) to explicitly considering calibration data and picking the most reliable set of qubits for the computation (Murali et al. 2019). However, in most cases, it is not possible to globally define a mapping which conforms to *all* connectivity limitations. As a consequence, the logical-to-physical qubit mapping is usually changed dynamically throughout the circuit. Typically, this is accomplished by inserting *SWAP* gates into the circuit – effectively allowing to change the mapping of logical qubits to physical qubits so that all operations can be executed while, at the same time, all connectivity constraints are satisfied. Several approaches have been proposed for tackling this immensely complex task (In fact, the mapping task has been shown to be NP-complete (Siraichi et al. 2018).) (Zulehner et al. 2019d; Smith and Thornton 2019; Wille et al. 2019; Li et al. 2019; Matsuo et al. 2019; Murali et al. 2019; Siraichi et al. 2018; Amy and Gheorghiu 2019; Zulehner and Wille 2019a; Burgholzer et al. 2022a).

Example 3. Consider again the circuit G from Example 1, and assume that the Toffoli gate has been synthesized as shown in Fig. 1b. Further, assume that the circuit is to be executed on the IBMQ London architecture shown in Fig. 1c. Then, Fig. 1d shows one possible circuit $\tilde{G}$ resulting from this mapping process. The physical qubits Q_0, Q_1, and Q_2 were chosen and initially assigned logical qubits q_0, q_2, and q_1, respectively. Just one *SWAP* operation applied to Q_0 and Q_1 (indicated by $\times$) was added in the middle of the circuit in order to conform to the target's connectivity constraints (A *SWAP* operation is eventually realized using three *CNOT* operations as indicated in the middle of Fig. 1.).

After this step of the compilation flow, circuits are ready to be executed on the targeted devices. However, the previous steps significantly increased the size of these circuits – impacting the achievable performance due to the limited coherence time and gate fidelity. Thus, several **optimizations** may be employed to reduce the circuit's size and, hence, improve the actual performance on the

quantum computer. This might include fusing consecutive gates acting on the same qubits, cancelling adjacent gates that are inverse to another (e.g., two consecutive *CNOT* operations with the same control and target qubits), or more sophisticated optimization techniques such as gate transformation and commutation (Itoko et al. 2020) or resynthesis of two-qubit unitary blocks (Vidal and Dawson 2004).

Example 4. Consider again the circuit $\tilde{G}$ from Example 3 shown in Fig. 1d that has been mapped to the IBMQ London architecture. Applying one-qubit fusion and adjacent-gate cancellation eventually allows to eliminate nine single-qubit gates and results in the *optimized* circuit G' shown in Fig. 1e.

Verification

Naturally, it is of utmost importance that, after compilation, the resulting (compiled) circuit still implements the same functionality as the originally given circuit. This can be guaranteed by *verifying* the results of the compilation flow, i.e., checking the equivalence of the original circuit description with the compiled quantum circuit – an established task in classical computing. A similar take can be used for quantum computing. However, the special characteristics of quantum circuits need to be accounted for, while, at the same time, exploiting additional potential wherever possible. This section reviews how verification for classical circuits is conducted and how this translates to the domain of quantum computing. Based on that, the remainder of this chapter discusses how a corresponding verification flow for quantum circuits can be established.

Classical Circuits

In order to demonstrate or even prove the correctness of classical circuits, verification methods are applied. They check whether a given circuit, the *design under verification* (DUV), adheres to an also given *golden specification*. To this end, current (industrial) practice mainly applies schemes such as:

- *Formal verification* (Biere and Kunz 2002; Drechsler 2004), which considers the problem mathematically and proves that a circuit is correct with 100% certainty
- *Simulation-based verification* (Yuan et al. 2006; Bergeron 2006; Kitchen and Kuehlmann 2007; Wille et al. 2009; Le et al. 2019; Laeufer et al. 2018), in which certain input assignments (*stimuli*) are explicitly assigned to the circuit and propagated through it and the outputs are compared to the expected values

Obviously, formal verification provides the best solution with respect to quality. The corresponding methods are capable of efficiently traversing large parts of the search space, e.g., by applying clever implications during the proof. The corresponding techniques are, however, rather complex compared to their simulation-based

counterparts and, particularly for larger designs, often fail due to the exponential complexity of the task.

Simulation is much easier to implement and very fast as long as only a limited number of stimuli is applied. The problem obviously is the quality provided by the applied set of stimuli. An exhaustive set of stimuli would show correctness with 100% certainty but is practically intractable as this would eventually require an exponential number of stimuli to simulate. Accordingly, methods such as *constraint-based random simulation* (Yuan et al. 2006; Bergeron 2006; Kitchen and Kuehlmann 2007; Wille et al. 2009), *fuzzing* (Le et al. 2019; Laeufer et al. 2018), etc. are key techniques to cope with this problem, while still maintaining a high quality. Here, stimuli and/or data inputs are specifically generated (e.g., from constraints, mutations of randomly generated inputs, etc.) so that corner case scenarios and/or a broad variety of cases are triggered. In doing so, errors that might otherwise remain undetected are more likely to be found.

However, despite substantial progress that has been made in the past, e.g., on improving the efficiency of formal methods or on stimuli generation which increases the coverage of simulative verification, verifying classical circuits remains a challenge and, hence, is subject of further research.

Quantum Circuits

In the quantum realm, the verification problem can be stated in a similar fashion as for classical circuits: given a circuit G, which acts as a *specification* (potentially consisting of high-level operations or building blocks), and a quantum circuit G', which acts as an *implementation*, it shall be checked whether the implementation adheres to the specification (Note that the terms *device under verification* and *golden specification* are not established in the quantum realm (yet), which is why, in the following, two quantum circuits G and G' are considered that act as *specification* and *implementation*.). More specifically, given two quantum circuits $G = g_0 \dots g_{|G|-1}$ and $G' = g'_0 \dots g'_{|G'|-1}$ with corresponding system matrices U and U', the *equivalence checking problem for quantum circuits* asks whether:

$$U = e^{i\theta} U' \text{ or, equivalently, } U U'^{-1} = e^{i\theta} I,$$

where $\theta \in (-\pi, \pi]$ denotes a physically unobservable global phase (Since the probability of measuring the basis state $|i\rangle$ from an n-qubit quantum state $|\varphi\rangle$ is given by $|\alpha_i|^2$ and $|e^{i\theta}|^2 = 1$, the states $|\varphi\rangle$ and $e^{i\theta}|\varphi\rangle$ produce the same measurement probability distributions and, hence, are physically indistinguishable.). So, in principle, checking the equivalence of two quantum circuits reduces to the comparison of the respective system matrices U and U'.

Example 5. Consider again the circuits G and G' shown in Fig. 1a and e, respectively. Then, after accounting for the initial layout and the output permutation of

G' (explained in more detail later in section "Designing a Strategy for Verifying Compilation Flow Results"), both circuits realize the same unitary matrix:

$$U = -\frac{1}{\sqrt{2}} \begin{bmatrix} 0 & 0 & 0 & -1 & \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} \\ 0 & 1 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 & 0 & \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ 1 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} \\ 0 & 0 & 0 & 1 & \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} \\ 0 & -1 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & -1 & 0 & \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ -1 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} \end{bmatrix}.$$

Thus, both circuits are considered equivalent.

If U and U' differ in any column i (by more than a global phase factor $e^{i\theta}$), then the corresponding circuits G and G' are not equivalent, and $|i\rangle$ serves as a counterexample showing that:

$$U|i\rangle = |u_i\rangle \not\equiv |u_i'\rangle = U'|i\rangle.$$

Here, the fidelity $\mathcal{F}$ between two states $|x\rangle$ and $|y\rangle$ is typically used as a similarity measure for comparing quantum states, where $\mathcal{F}$ is calculated as the squared overlap between the states, i.e., $\mathcal{F} = |\langle x|y\rangle|^2 \in [0, 1]$. Two states are considered equivalent if the fidelity between them is 1 (up to a given tolerance ε).

Example 6. Consider the same scenario as in Example 5, but additionally assume that, due to an error, the first X gate of G' is not applied (yielding a new circuit $\tilde{G}'$). Then, a new functionality results, which is described by the system matrix:

$$\tilde{U}' = -\frac{1}{\sqrt{2}} \begin{bmatrix} -\frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} & \frac{1}{2} & -1 & 0 & 0 & 0 \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 1 & 0 \\ \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} & 0 & 1 & 0 & 0 \\ -\frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 1 \\ -\frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} & \frac{1}{2} & 1 & 0 & 0 & 0 \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} & \frac{1}{2} & 0 & 0 & -1 & 0 \\ \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} & 0 & -1 & 0 & 0 \\ -\frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & -1 \end{bmatrix}.$$

Since U and $\tilde{U}'$ are obviously not identical anymore, the circuits G and $\tilde{G}'$ have been shown to be nonequivalent. Moreover, since U and $\tilde{U}'$ differ in *all* of their columns, *any* computational basis state $|i\rangle$ serves as a counterexample, i.e.,

$\mathcal{F}(U|i\rangle, \tilde{U}'|i\rangle) < 1$ for all i from 0 to $2^n - 1$. This characteristic of quantum computing that even small errors frequently affect most (if not all) of a circuit's functionality will be further explored later in section "Simulative Verification".

Unfortunately, the whole functionality U (and similarly U') is not readily available for performing this comparison but has to be constructed from the individual gate descriptions g_i – requiring the subsequent matrix-matrix multiplications:

$$U^{(0)} = U_0, \quad U^{(j)} = U_j \cdot U^{(j-1)} \text{ for } j = 1, \ldots, |G| - 1$$

to construct the whole system matrix $U = U^{(|G|-1)}$. While conceptually simple, this quickly constitutes an extremely complex task due to the exponential size of the involved matrices with respect to the number of qubits. In fact, equivalence checking of quantum circuits has been shown to be QMA-complete (Janzing et al. 2005). In the following, two complementary approaches addressing this problem are presented. First, a formal verification approach is shown (in section "Formal Verification") that capitalizes on the inherent reversibility of quantum circuits. Then, the power of simulation for the verification of quantum circuits is demonstrated (in section "Simulative Verification"). Afterward, the composition of both complementary techniques into the first advanced quantum circuit equivalence checking flow is described (in section "Resulting Quantum Circuit Equivalence Checking Flow").

Formal Verification

As shown in the previous section, verification of quantum circuits by constructing and comparing their system matrices is infeasible in general due to the exponential growth of the matrices' dimensions with respect to the number of qubits. However, the respective matrices frequently are sparse or have inherent redundancies in their representation. *Decision diagrams* (Chin-Yung et al. 2011; Zulehner et al. 2019e; Viamontes et al. 2004; Niemann et al. 2016) have been proposed as a means to exploit these redundancies in the underlying representation, which allows them to compactly represent and efficiently manipulate quantum states and operations in many cases – providing the basis for an advanced formal verification approach. Moreover, the inherent reversibility of quantum computations allows for more efficiently checking the equivalence of two quantum circuits. This section covers that by first reviewing decision diagrams and the corresponding basic verification approach (based on Niemann et al. 2016). Afterward, an *alternating* scheme that takes advantage of the mentioned characteristics is described (based on Burgholzer and Wille 2020a), and it is shown how a corresponding strategy that is specifically tailored for verifying compilation results can be designed (based on Burgholzer et al. 2020).

Decision Diagrams

For our purposes, a decision diagram is a directed, acyclic graph with complex edge weights. Consider a unitary matrix $U \in \mathbb{C}^{2^n \times 2^n}$. Then, U can be split into four $2^{n-1} \times 2^{n-1}$-sized sub-matrices U_{ij} as shown on the left side of Fig. 2. This splitting corresponds to the action of U depending on the value of the topmost qubit q_{n-1}, i.e., U_{ij} describes how the rest of the system is transformed given that q_{n-1} is mapped from $|j\rangle$ to $|i\rangle$ for $i, j \in \{0, 1\}$. In the corresponding decision diagram, this manifests as a node with label $n-1$ and four successor nodes as shown on the right side of Fig. 2.

This decomposition scheme can now be applied recursively until only single matrix entries (i.e., complex numbers) remain. The resulting structure has n levels of nodes, labeled $n-1$ down to 0. Sub-matrices only differing by a constant factor can be represented by the same node in the decision diagram. In general, this is handled by employing normalization schemes that guarantee canonicity and using hash tables to track unique nodes. The corresponding common factors are stored as edge weights in the diagram. In this fashion, rather compact representations for many functional descriptions of quantum algorithms can be obtained.

Example 7. Consider again the circuit G shown in Fig. 1a and its corresponding unitary matrix U shown in Example 5. Then, Fig. 3 shows the corresponding decision diagram. To this end, the decision diagram visualization method proposed in Wille et al. (2021) is adopted, where thickness and color of an edge represent the

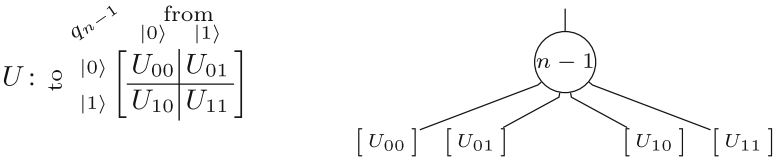


Fig. 2 Decomposition scheme

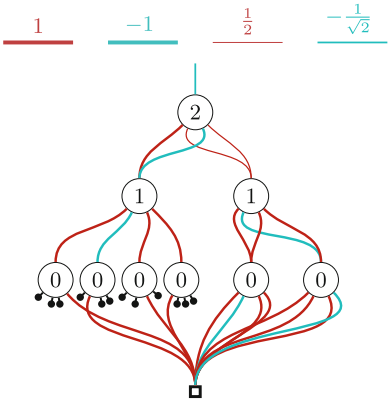


Fig. 3 Decision diagram for the functionality of the circuit G from Fig. 1a

edge weight's magnitude and phase, respectively. Obviously, the decision diagram representation is much more compact than the whole matrix.

Furthermore, most operations on vectors and matrices – multiplication, addition, inner/outer product, tensor product, etc. – can naturally be translated to decision diagrams due to their recursive nature. However, instead of scaling with the vectors' and matrices' dimensions, operations on decision diagrams scale with the number of nodes of the respective decision diagrams. Hence, as long as the involved decision diagrams remain compact, computations can be conducted very efficiently.

Example 8. The multiplication of two matrices U and U' can be recursively broken down according to the following:

$$\begin{bmatrix} U_{00} & U_{01} \\ U_{10} & U_{11} \end{bmatrix} \times \begin{bmatrix} U'_{00} & U'_{01} \\ U'_{10} & U'_{11} \end{bmatrix} = \begin{bmatrix} (U_{00} \cdot U'_{00} + U_{01} \cdot U'_{10}) & (U_{00} \cdot U'_{01} + U_{01} \cdot U'_{11}) \\ (U_{10} \cdot U'_{00} + U_{11} \cdot U'_{10}) & (U_{10} \cdot U'_{01} + U_{11} \cdot U'_{11}) \end{bmatrix},$$

with $U_{ij}^{(\prime)} \in \mathbb{C}^{2^{n-1} \times 2^{n-1}}$ for $i, j \in \{0, 1\}$. In the respective decision diagrams, U_{ij} and U'_{ij} directly correspond to the successors of a node. As a consequence, the complexity of the multiplication scales with the product of the respective number of nodes.

General Approach

Decision diagrams are predestined for verification, because they are canonical (with respect to a particular variable order and normalization criterion), i.e., there are no two different decision diagrams for the same functionality. Once the decision diagrams for both circuits G and G' in question are constructed, e.g., using the techniques proposed in Burgholzer et al. (2021a), it suffices to compare their root pointers and the corresponding top edge weight (Niemann et al. 2016).

Example 9. Consider again the scenario as in Example 5. Then, constructing the decision diagrams for both circuits in either case results in the diagram shown in Fig. 3. Their equivalence can be concluded by comparing the respective root pointers and top edge weights (which are identical).

While this is true in theory, numerical inaccuracies present a notorious practical issue. Because quantum gates are described by matrices over $\mathbb{C}$, they are hard to accurately represent in memory. Usually, these matrices are stored using floating point numbers which lead to imprecision and rounding errors. As a consequence, two decision diagrams might not be exactly identical despite being equivalent in theory. Thus, simply comparing the root pointers of the resulting decision diagrams is not enough to determine the equivalence of two circuits in practice. Instead, the Hilbert-Schmidt inner product can be used to quantify the similarity between

two matrices (and, hence, decision diagrams). Let tr denote the trace of a matrix, i.e., the sum of its diagonal elements. Then, because $\mathrm{tr}(I) = 2^n$ for the identity transformation on n qubits, one can check whether $|\mathrm{tr}(UU'^{-1})| \approx 2^n$ in order to conclude the equivalence of both circuits up to a given tolerance. This requires further, potentially expensive, operations.

Furthermore, while decision diagrams frequently allow to compactly represent the functionality of a quantum circuit, their worst-case complexity is still exponential. Hence, it might not be possible to efficiently construct a representation of a circuit's full functionality. But, as mentioned above, characteristics of quantum computing, specifically its inherent reversibility, offer promising potential for a complementary approach. This is covered next.

Alternating Approach

If G and G' are equivalent, then it holds that $G'^{-1}G = I$, i.e., concatenating one circuit with the inverse of the other implements the identity. Since the identity has a perfectly compact representation as a decision diagram, being linear in the number of qubits (as shown in Fig. 4), the decision diagram for the combined circuit $G'^{-1}G$ can be constructed instead of constructing the individual circuits' decision diagrams. More specifically, this entails the computation of the following:

$$G'^{-1}G \equiv U \cdot U'^{-1} = U_{|G|-1} \cdots U_0 \cdot U_0'^{-1} \cdots U_{|G'|-1}'^{-1}.$$

However, building up the decision diagram of $G'^{-1}G$ sequentially from left to right might still result in an exponentially large decision diagram, since eventually the whole decision diagram for G is constructed in the middle of the computation.

Hence, the much better solution (proposed in Burgholzer and Wille 2020a) is to start constructing the functionality of the combined circuit "from the middle" and *alternate* between applications of gates from G ("from the left") and inverted gates from G' ("from the right"), i.e.:

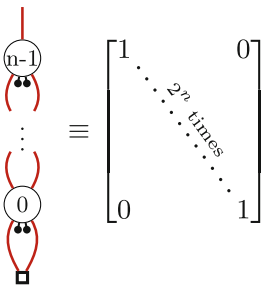


Fig. 4 Decision diagram of the n-qubit identity matrix

$$\begin{aligned} I = G'^{-1} \cdot G &= (g'^{-1}_{m'-1} \dots g'^{-1}_0) \cdot (g_0 \dots g_{m-1}) \\ &\equiv (U_{m-1} \cdots U_0) \cdot (U'^{\dagger}_0 \cdots U'^{\dagger}_{m'-1}) \\ &= U_{m-1} \cdots U_0 \cdot I \cdot U'^{\dagger}_0 \cdots U'^{\dagger}_{m'-1} \\ &=: G \to I \leftarrow G'. \end{aligned}$$

The intention of this idea is to keep the intermediate computations as close to the identity as possible, since the identity constitutes the best case for most representations of quantum functionality (e.g., linear in the number of nodes with respect to the number of qubits for decision diagrams).

Example 10. Assume, without loss of generality, that $m \leq m'$, i.e., G' has at least as many gates as G. Further assume an *oracle* $\omega \colon G \to (G')^*$ exists that, given a gate $g_i \in G$, returns a consecutive sequence of gates $g'_k \dots g'_l \in G'$ such that $g_i \equiv g'_k \dots g'_l$. Then, subsequently applying one gate $g \in G$ and $|\omega(g)|$ inverted gates from G' constitutes a "perfect" strategy – yielding the identity after each pair of applications. As a result, only matrices representing, or staying close to, the identity occur. Since these can usually be represented very efficiently using, e.g., decision diagrams, the process of equivalence checking is substantially improved. An easily accessible online tool (based on Wille et al. 2021), where the execution of gates "from the left" or "from the right" can be nicely explored, is available at iic.jku.at/eda/research/quantum_dd/tool/.

This method also makes it easier to check equivalence of circuits up to some precision using the inner product $\mathrm{tr}(UU'^{-1})$, since the product UU'^{-1} is inherently constructed during the equivalence check – saving a potentially expensive decision diagram multiplication. However, a major problem remains in how to obtain the "perfect" oracle $\omega(\cdot)$, i.e., in deciding when to apply operations of G ("from the left") and when to apply operations of G' ("from the right"). Here, information on the origin of the two circuits to be checked may help, e.g., information from the compilation flow when trying to verify compilation results, which is discussed next.

Designing a Strategy for Verifying Compilation Flow Results

The compilation flow as reviewed in section "Quantum Circuit Compilation" provides detailed insights on how a circuit G is eventually compiled to a circuit G' – providing ideal knowledge about how to derive the "perfect" oracle $\omega(\cdot)$ (Burgholzer et al. 2020).

Considering the *synthesis* step of the compilation flow, two issues become relevant for determining the "perfect" $G \to \mathbb{I} \leftarrow G'$ strategy: (1) each gate $g \in G$ is compiled to a sequence of gates $g'_k \dots g'_l \in G'$, and (2) the circuits G and G' may operate on different numbers of qubits due to the addition of ancillary qubits required for the synthesis.

For the first issue, it can be exploited that the actual decomposition scheme, i.e., into how many elementary gates each of the original circuit's gates is decomposed, is known a priori. Thus, an *oracle* $\omega(\cdot)$ which, given a gate $g \in G$, returns the corresponding sequence of gates $g'_k \dots g'_l \in G'$ is explicitly known in this case. Assuming that G' resulted from the synthesis of a given quantum circuit G, applying *one* gate from G and $|\omega(g)|$ inverted gates from G' constitutes an optimal strategy for conducting $G \to \mathbb{I} \leftarrow G'$ – yielding the identity after each step.

Example 11. Consider the original circuit G shown in Fig. 1a. As indicated by Fig. 1b, the Toffoli gate of G needs to be decomposed into elementary gates supported by the architecture, while all other gates of G are already supported. Thus, $|\omega(g)| = 1$ holds for all $g \in G$ except for the Toffoli gate, where $|\omega(g)| = 15$ holds.

In case both circuits do not operate on the same number of qubits, the corresponding unitaries have different dimensions and cannot be applied directly. Unfortunately, it is not sufficient to match the qubit count of G' by just augmenting the original circuit with idle qubits. Since ancillary qubits are always initialized in a particular state (typically $|0\rangle$), this leaves some degree of freedom in the overall unitary representation U'. In order to compensate for this degree of freedom, the eventually resulting matrix U' has to be modified as shown in the following example.

Example 12. Consider a unitary $2^n \times 2^n$ matrix U, and assume that, w.l.o.g., the last qubit q_{n-1} acts as an ancillary qubit initialized to $|0\rangle$. In general, the action of U depending on the state of q_{n-1} is described by the four $2^{n-1} \times 2^{n-1}$ sub-matrices U_{ij} as illustrated in Fig. 5a. Since the ancillary is initialized to $|0\rangle$, the sub-matrices corresponding to the transformation from $|1\rangle$ can be ignored – resulting in the *modified matrix* $\tilde{U}$ shown in Fig. 5b.

Considering the *mapping* step of the compilation flow, it becomes an issue that a connection between the circuit's logical and the device's physical qubits is established. Consequently, while the description of G is expressed in terms of logical qubits $q_0, \dots, q_{n-1}$, the circuit G' operates on (a subset of) the device's physical qubits $Q_0, \dots, Q_{N-1}$. If a nontrivial initial mapping (i.e., anything but

(a) Original matrix U (b) Modified matrix $\tilde{U}$

Fig. 5 Handling of ancillary qubits. (**a**) Original matrix U. (**b**) Modified matrix $\tilde{U}$

$q_i \mapsto Q_i$) is employed, this leads to the situation that gates from G', although functionally equivalent, are applied to different qubits than the gates of G. Thus, concluding the equivalence of both circuits is not possible by straightforwardly using the oracle function $\omega(\cdot)$. Instead, a *qubit map* $m(\cdot)$ is employed, which stores the mapping between the physical qubits of the circuit G' and the logical qubits of the original circuit G, i.e., $m(Q_i) = q_j$ if physical qubit Q_i is initially assigned logical qubit q_j. Whenever a gate from G' is to be applied to a certain physical qubit Q_i, this is translated to the corresponding logical qubit $m(Q_i) = q_j$ – again allowing to stay close to the identity.

Example 13. Consider the original circuit G and the mapped circuit $\tilde{G}$ shown in Fig. 1a and d, respectively. While the X gate at the beginning of G is applied to the logical qubit q_2, it is applied to the physical qubit Q_1 in the circuit $\tilde{G}$. In order to fix this mismatch, the qubit map $m(\cdot)$ – mapping $Q_0 \mapsto q_0$, $Q_1 \mapsto q_2$, and $Q_2 \mapsto q_1$ – is employed. Consequently, the X gate of $\tilde{G}$ is applied to $m(Q_1) = q_2$ which now matches the original gate from G perfectly.

However, as discussed in section "Quantum Circuit Compilation", the logical-to-physical qubit mapping of a compiled circuit in general changes dynamically throughout the circuit in order to satisfy all constraints imposed by the device's coupling map. As a consequence, the potential of using the (static) qubit map $m(\cdot)$ in combination with the oracle function $\omega(\cdot)$ to stay close to the identity is significantly diminished, that is, because the dynamically changed mapping again results in a scenario where gates from G' are applied to different qubits than in the circuit G. Therefore, a perfect verification strategy needs to keep track of the changes in the logical-to-physical qubit mapping caused by *SWAP* operations (*SWAP*s can be reconstructed from consecutive sequences of three *CNOT*s in G' as indicated in the middle of Fig. 1.) and, accordingly, needs to *update the qubit map* $m(\cdot)$ throughout the verification procedure.

Example 14. Consider again the scenario of Example 13. If the $G \rightarrow \mathbb{I} \leftarrow G'$ scheme is carried out using the qubit map $m(\cdot)$ defined there, the result would not represent the identity. That is, because the logical-to-physical qubit mapping is changed in the middle of $\tilde{G}$ by a *SWAP* operation applied to Q_0 and Q_1. Thus, at that specific point, the qubit map $m(\cdot)$ has to be updated accordingly, i.e., it then has to map $Q_0 \mapsto q_2$, $Q_1 \mapsto q_0$, and $Q_2 \mapsto q_1$. Through this dynamic change, the computation of $G \rightarrow \mathbb{I} \leftarrow G'$ remains close to the identity and, eventually, proves the equivalence of both circuits.

If no optimizations were to be applied to the circuit resulting from the synthesis and mapping step, the strategies proposed above allow to conduct $G \rightarrow \mathbb{I} \leftarrow G'$ in a perfect fashion – yielding the identity after each step. However, optimizations further alter the circuit – making it harder to verify the resulting circuit. In the following, it is demonstrated how such optimizations can be accounted for by using *single-qubit gate fusion* as an example.

Example 15. Consider again the circuit $\tilde{G}$ shown in Fig. 1d. There, the gray box indicates the gates of $\tilde{G}$ realizing the Toffoli gate of the original circuit G shown in Fig. 1a. The middle qubit thereby contains a T gate, which is directly followed by an H gate. Accordingly, in the optimized circuit shown in Fig. 1e, these have been merged into a single $U_2(0, \frac{5\pi}{4})$ gate. Thus, $|\omega(g)| = 15$ does no longer hold but has to be modified to $|\omega(g)| = 14$ instead in case of the Toffoli gate (see Example 11).

In addition to anticipating fusions within individual gate realizations through adaptations of the oracle function $\omega(\cdot)$, a *preprocessing* pass is conducted which fuses consecutive single-qubit gates where they are present in the original circuit G (e.g., fusing the $H - X - H$ cascade at the end of the circuit G shown in Fig. 1a to a single Z gate). However, reductions across multiple gates that were decomposed during synthesis cannot be accounted for in this fashion. Thus, the formerly constructed *perfect* oracle function $\omega(\cdot)$ becomes *approximate*.

Example 16. Consider again the circuit $\tilde{G}$ shown in Fig. 1d and its optimized variant G' shown in Fig. 1e. Then, the cancellation of the two consecutive H gates in the beginning of $\tilde{G}$ cannot be anticipated through a straightforward adaptation of $\omega(\cdot)$. However, $\omega(\cdot)$ remains a suitable approximation for staying close to the identity.

All of the considerations above finally result in a dedicated formal verification strategy that is tailored for verifying results of compilation flows.

Simulative Verification

The formal verification techniques presented in the previous section allow to efficiently check the equivalence of two quantum circuits in many cases. But, at the same time, the inherent reversibility of quantum computations motivates another complementary solution to the verification problem, namely, *simulation*. In contrast to classical circuits and systems, where errors are frequently masked due to the inherent information loss introduced by many logic gates, even small errors in general affect most (if not all) the functionality of a quantum system. As a consequence, in order to check the equivalence of two quantum circuits, it might not be necessary to consider their complete functionality. Instead, the simulation of both computations with a couple of arbitrary input states, i.e., considering only a small part of the whole functionality, reliably allows for the detection of errors in many cases and, hence, provides an attractive alternative to formal verification as described above. This section (which is based on Burgholzer and Wille 2020b; Burgholzer et al. 2021b) covers this by first reviewing what comprises such methods and, afterward, describing different stimuli generation schemes that offer a trade-off between expressiveness and efficiency.

Verification Schemes Based on Simulation

While the construction of the matrix U (and accordingly of U') requires expensive matrix-matrix multiplications, the simulation of G with input state $|\varphi\rangle$ only entails the matrix-vector multiplications:

$$|\varphi^{(0)}\rangle = U_0 |\varphi\rangle , \quad |\varphi^{(j)}\rangle = U_j \cdot |\varphi^{(j-1)}\rangle \text{ for } j \in \{1, \ldots, m-1\}.$$

If simulating both circuits with the same input yields different outputs, the circuits have been shown to be *nonequivalent*. This constitutes an exponentially easier task than constructing the entire system matrices U and U' – although the complexity of simulation still remains exponential with respect to the number of qubits. Accordingly, verification based on simulations of the circuits in question might provide a promising alternative. In fact, this has already been considered in theoretical quantum information, where (truly quantum-based) methods have been proposed (see e.g., Watrous 2018, Section 3 and Khatri et al. 2019). But these approaches would require an execution on actual quantum computing devices, whose availability and accessibility still are severely restricted. Hence, before valuable quantum computing resources are wasted to verify a quantum circuit, efficient alternatives which can be employed prior to an actual execution on a quantum computer (using classical computing devices) are of high interest (This has similarities to the verification of classical circuits which also shall be conducted prior to an actual execution in the field.).

Such a quantum circuit verification scheme based on simulation can, in general, be described as follows:

1. Consider a set $\mathcal{S}$ of quantum states (which serve as stimuli).
2. Pick (and prepare) a quantum state $|\varphi\rangle \in \mathcal{S}$.
3. Simulate (on a classical device) both G and G' with this initial state – resulting in two states $|\varphi_G\rangle$ and $|\varphi_{G'}\rangle$, respectively.
4. Compare the output $|\varphi_{G'}\rangle$ generated by the implementation G' with the desired output $|\varphi_G\rangle$ by computing the quantum fidelity $\mathcal{F}$ between both states (Nielsen and Chuang 2010), i.e.:

$$\mathcal{F}(|\varphi_G\rangle , |\varphi_{G'}\rangle) = |\langle\varphi_G\rangle \varphi_{G'}|^2 \in [0, 1].$$

5. If $\mathcal{F}(|\varphi_G\rangle , |\varphi_{G'}\rangle) \neq 1$, the stimulus $|\varphi\rangle$ shows the incorrect behavior of G' with respect to G. Accordingly, the verification failed and the process is terminated.
6. Remove $|\varphi\rangle$ from $\mathcal{S}$.
7. If $|\mathcal{S}| \neq \emptyset$ (i.e., $\mathcal{S}$ is still non-empty), continue with Step (2); otherwise, the verification process has been completed.

Now, the challenges of such an approach are as follows: First, simulating a quantum circuit $G = g_0 \ldots g_{|G|-1}$ starting with an initial state $|\varphi\rangle$ on a classical device (Step (3) from above) is substantially harder than for the verification of classical

circuits (here, a single simulation yields only linear complexity). However, rather powerful methods have been proposed to tackle this complexity – including methods based on highly optimized and parallel matrix computations (Guerreschi et al. 2020; Jones et al. 2018; Doi et al. 2019), tensor networks (Villalonga et al. 2019; Pednault et al. 2019; Vincent et al. 2021; Brennan et al. 2021), quasiprobability/stabilizer-rank methods (Seddon et al. 2020, and references therein), as well as decision diagrams (Zulehner and Wille 2019b,c; Burgholzer et al. 2021c, 2022b).

Second, as in the verification of classical circuits, the quality of the verification process heavily depends on the applied set of stimuli, i.e., 100% certainty cannot be guaranteed as long as the set of applied stimuli is not exhaustive. Moreover, while the stimuli space for classical circuits is finite (each input bit can be assigned either 0 or 1 – yielding a total of 2^n possible stimuli), the state space in the quantum realm is infinitely large (possible stimuli are elements of a 2^n-dimensional Hilbert space).

Although the perspective of a possible infinite number of stimuli may seem rather grim at a first glance, there are promising ways to check the correctness of quantum circuits using simulative verification. These, however, severely depend on how the stimuli are actually generated. In fact, high error detection rates can be achieved even if only a few randomly chosen stimuli are considered – as long as these are generated in a specific fashion. Corresponding schemes offering a trade-off between expressiveness and efficiency are covered next.

Stimuli Generation Schemes

In the following, different schemes for the generation of (random) stimuli are illustrated, and it is explored how well they can show the correctness of a quantum circuit.

The most straightforward application of simulative verification for quantum circuits is to consider the set of computational basis states as stimuli (i.e., picking $|\varphi\rangle$ from the set $\{|i\rangle : i \in \{0,1\}^n\}$) and computing $\mathcal{F}(U|i\rangle, U'|i\rangle)$.

Example 17. Consider an n-qubit quantum circuit G, and assume that some error affects (w.l.o.g.) the first qubit in the actual realization G'. In the quantum realm, this means that the circuit G' is described by the unitary matrix:

$$U' = (\mathbb{I}^{\otimes(n-1)} \otimes E) \cdot U,$$

where E describes an error gate that is applied to the first qubit. Due to the inherent reversibility of quantum gates, this error has a localized effect on the output, i.e.:

$$\mathcal{F}(U|c\rangle, U'|c\rangle) = \mathcal{F}(|c\rangle, (\mathbb{I}^{\otimes(n-1)} \otimes E)|c\rangle) = |\langle c_0|E|c_0\rangle|^2$$

for any classical stimulus $|c\rangle = |c_{n-1} \dots c_0\rangle$.

Now suppose that $E = X$, i.e., a bit flip error occurred. In contrast to classical intuition, such an error can be detected by a *single* simulation with *any* classical stimulus $|c\rangle$, since $\mathcal{F}(U|c\rangle, U'|c\rangle) = |\langle c_0|X|c_0\rangle|^2 = 0$ independent of $|c\rangle$.

However, this approach has a severe handicap – namely, that it is not faithful. Specifically, for each quantum circuit G, there is an (infinitely large) family of realizations G' for which $\mathcal{F}(U\,|c\rangle\,,U'\,|c\rangle) = 1$ holds for all classical stimuli $|c\rangle$, even if quantum states $|\varphi\rangle$ with $\mathcal{F}(U\,|\varphi\rangle\,,U'\,|\varphi\rangle) \neq 1$ actually exist. An example illustrates the problem:

Example 18. Consider the same scenario as in Example 17, but assume that the error is characterized as $E = Z$, i.e., a phase flip error occurred. No classical stimulus $|c\rangle$ may detect such an error due to the fact that $\mathcal{F}(U|c\rangle, U'|c\rangle) = |\langle c_0|Z|c_0\rangle|^2 = 1$ independent of $|c\rangle$. Intuitively, this happens whenever the "difference" of U and U' is diagonal in the computational basis, such as $\mathbb{I}^{\otimes(n-1)} \otimes Z$ in case of this example.

Nevertheless, it has been shown that whenever classical stimuli are actually capable of detecting a certain error in the realization G', they do so within remarkably few simulations with randomly picked classical stimuli – an effect contradictory to classical intuition.

The fact that classical stimuli generation is not sufficient to faithfully detect errors in quantum circuits should not come as a surprise. After all, quantum circuits are designed to achieve tasks that classical circuits cannot. In fact, a closer look at the single-(qu)bit case already reveals a fundamental discrepancy: classical single-bit operations map one of two possible inputs (0 or 1) to one of two possible outputs (0 or 1). In contrast, the quantum case is much more expressive: the set of all possible single-qubit states $|\varphi\rangle$ is infinitely large and can be parametrized by the two-dimensional Bloch sphere (Nielsen and Chuang 2010) illustrated in Fig. 6. Single-qubit quantum operations map single-qubit states to single-qubit states. Geometrically, this family encompasses all possible rotations of the Bloch sphere as well as all reflections. Classical (single-qubit) stimuli, i.e., the states $|0\rangle$ and $|1\rangle$, are not expressive enough to reliably probe such a continuum of operations. They correspond to antipodal points on the (Bloch) sphere, and it is simply impossible to detect certain transformations by tracking the movement of only two antipodal points.

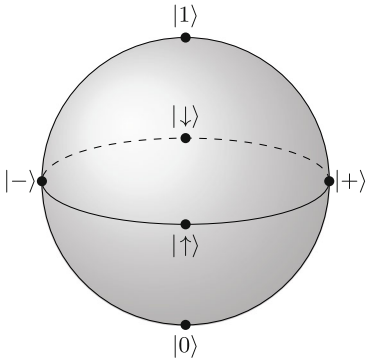


Fig. 6 Bloch sphere

In order to address this, also stimuli beyond (classical) basis states should be considered. More precisely, three pairs of antipodal points are sufficient for full resolution (Schwinger 1960; Klappenecker and Rotteler 2005; Kueng and Gross 2015), namely:

$$
\begin{aligned}
&|0\rangle\,, && |1\rangle\,, && (Z-\text{basis}),\\
&|+\rangle = 1/\sqrt{2}(|0\rangle + |1\rangle), && |-\rangle = 1/\sqrt{2}(|0\rangle - |1\rangle), && (X-\text{basis}), \text{ and}\\
&|\uparrow\rangle = 1/\sqrt{2}(|0\rangle + i\,|1\rangle), && |\downarrow\rangle = 1/\sqrt{2}(|0\rangle - i\,|1\rangle), && (Y-\text{basis}).
\end{aligned}
$$

Generating stimuli uniformly at random from this sextuple (The single-qubit states $|0\rangle\,, |1\rangle\,, |+\rangle\,, |-\rangle\,, |\uparrow\rangle\,, |\downarrow\rangle$ can be generated from the basis state $|0\rangle$ by applying the gates $\mathbb{I}$, X, H, XH, HS, or XHS, respectively.) produces a set that is expressive enough to detect *any* single-qubit error. More precisely, for any pair of functionally different single-qubit unitaries U and U', at least one input $|l_1\rangle \in \{|0\rangle\,, |1\rangle\,, |+\rangle\,, |-\rangle\,, |\uparrow\rangle\,, |\downarrow\rangle\}$ produces functionally different outputs, i.e., the fidelity $\mathcal{F}(U\,|l_1\rangle\,, U'\,|l_1\rangle)$ is guaranteed to be $\neq 1$.

This desirable feature extends to the multi-qubit case. That is, if one of these six (single-qubit) states for every available qubit is independently selected, every "local" single-qubit error may be detected. Thus, for n qubits, consider the following ensemble of *local quantum stimuli*:

$$
|l\rangle = |l_{n-1}\rangle \otimes \cdots \otimes |l_0\rangle \text{ with } |l_i\rangle \in \{|0\rangle\,, |1\rangle\,, |+\rangle\,, |-\rangle\,, |\uparrow\rangle\,, |\downarrow\rangle\}
$$

Example 19. Let us revisit the scenario from Example 17 (and Example 18). Compared to classical stimuli, local quantum stimuli behave in a more homogeneous fashion on the classical extreme cases shown before: first, suppose that $E = X$ (bit flip error). Then:

$$
\mathcal{F}(U|l\rangle, V|l\rangle) = |\langle l_0|X|l_0\rangle|^2 = \begin{cases} 0 & |l_0\rangle \in \{|0\rangle, |1\rangle, |\uparrow, |\downarrow\rangle\} \\ 1 & |l_0\rangle \in \{|+\rangle, |-\rangle\} \end{cases}
$$

Compared to classical stimuli, only $2/3$ of all local quantum stimuli detect this type of error. Now, suppose that $E = Z$ (phase flip error). Then:

$$
\mathcal{F}(U|l\rangle, V|l\rangle) = |\langle l_0|Z|l_0\rangle|^2 = \begin{cases} 0 & |l_0\rangle \in \{|+\rangle, |-\rangle, |\uparrow, |\downarrow\rangle\} \\ 1 & |l_0\rangle \in \{|0\rangle, |1\rangle\} \end{cases}
$$

Consequently, in contrast to not detecting such an error with classical stimuli at all, again $2/3$ of all local quantum stimuli are capable of detecting this type of error.

This observation that local quantum stimuli can detect errors which would have remained undetected using classical stimuli is not a coincidence. In fact, the collection of a total of 6^n local quantum stimuli is expressive enough to

detect *any* error in a quantum circuit. The key idea is to relate the expected fidelity $\mathbb{E}_{|l\rangle}\mathcal{F}(U\,|l\rangle\,,U'\,|l\rangle)$ – where the average is taken over all 6^n locally random stimuli – to a meaningful distance measure in the space of unitary matrices. This average outcome fidelity equals 1 if and only if U and U' are functionally equivalent. Now, suppose that U and U' are functionally distinct unitaries. Then, $\mathbb{E}_{|l\rangle}\mathcal{F}(U\,|l\rangle\,,U'\,|l\rangle) < 1$ which is only possible if (at least) one stimulus $|l\rangle$ produces an outcome fidelity that is strictly smaller than one. While this rigorous statement asserts that any error can be detected by (at least) one local quantum stimulus, it does not provide any advice on how to find the "right" stimulus. This is a very challenging problem in general, but the above example suggests that repeated random sampling of stimuli should "do the job." Typically, few randomly generated local quantum stimuli suffice to detect realistic errors.

As demonstrated above, a modest increase in the expressiveness of stimuli can already make a large difference. Local quantum stimuli can detect *any* error, while classical stimuli cannot. This is interesting, because local quantum stimuli are comparatively few in number (6^n states in a 2^n-dimensional state space to detect arbitrary discrepancies in unitary circuits) and actually do not inherit many further quantum features. For example, "global" quantum features such as entanglement are not employed by them at all. This begs the question: what kind of advantages can even more expressive and "more quantum" stimuli offer? Faithfulness is not a problem anymore, but richer, global stimuli may help to detect errors *earlier*, i.e., after substantially fewer iterations.

In order to identify powerful global quantum stimuli, it is helpful to revisit local quantum stimuli from a different perspective: they are generated through starting with a very simple classical state (i.e., $|0\ldots0\rangle$) and applying certain single-qubit gates to the individual qubits, e.g., $|0\rangle\otimes|+\rangle\otimes|\uparrow\rangle = (\mathbb{I}\otimes H\otimes HS)\,|000\rangle$. Consequently, *random* local stimuli are generated by choosing this *layer* of single-qubit gates at random. This generation scheme can be readily generalized. Rather than selecting only a single layer of (single-qubit) gates, a generation circuit $G_0\cdots G_{l-1}$ is constructed that has $l > 1$ layers and, most importantly, also features two-qubit gates. That is, a stimulus $|g\rangle$ with $|g\rangle = (G_0\cdots G_{l-1})\,|0\ldots0\rangle$ is generated, where each G_i is a (single) layer comprised of the so-called Clifford gates (H, S, *CNOT*) (Gottesman 1997).

Overall, this set of *global quantum stimuli* $|g\rangle$ contains all local quantum stimuli but is much richer and much more expressive. For instance, the overwhelming majority of global quantum stimuli will be highly entangled. Provided that the number of layers l is proportional to the number of qubits n (Hunter-Jones 2019; Brandão et al. 2016), these stimuli show remarkable properties. Most notably, the expected outcome fidelity (averaged over all possible global quantum stimuli $|g\rangle$) accurately approximates one of the most prominent distance measures for n-qubit quantum circuits, namely:

$$\mathbb{E}_{|g\rangle}\mathcal{F}(U|g\rangle, U'|g\rangle) \approx \mathcal{F}_{\text{avg}}(U,U') = \tfrac{1}{2^n+1}\left(1+2^n\left|\text{tr}(U^\dagger U')\right|^2\right).$$

A randomly selected global quantum stimulus $|g\rangle$ obeys the following:

$$\Pr_{|g\rangle}\left[\mathcal{F}(U|g\rangle, U'|g\rangle) = 1\right] \leq \mathcal{F}_{\text{avg}}(U, U').$$

The r.h.s. equals 1 if and only if G' correctly realizes G; otherwise it is typically *much* smaller. This general statement does have powerful implications when applied to a precise example.

Example 20. Consider again the scenario from Example 17 (and Example 18): a single-qubit error E occurred on the first qubit leading to the unitary:

$$U' = (\mathbb{I}^{\otimes(n-1)} \otimes E) \cdot U,$$

where the single-qubit error is either $E = X$ (bit flip error) or $E = Z$ (phase flip error). Then, $\mathcal{F}_{\text{avg}}(U, U') = \frac{1}{2^n+1} \leq 2^{-n}$ (because Pauli matrices are traceless) which implies that it is very unlikely to *not* detect this error with a single, random global quantum stimulus.

This example demonstrates the power of global quantum stimuli. However, it is important to keep in mind that this power is not for free. The generation of (random) global quantum stimuli and subsequent simulation is much more resource-intensive by comparison.

Resulting Quantum Circuit Equivalence Checking Flow

Eventually, this section (based on Burgholzer and Wille 2021a) describes how the individual ideas presented in the previous sections can be composed to form the first advanced equivalence checking flow for quantum circuits. Both techniques, formal and simulative verification, complement each other in many different ways. Trying to keep $G \rightarrow \mathbb{I} \leftarrow G'$ close to the identity proves very efficient in case two circuits are indeed equivalent – provided a "good" strategy can be employed. Conducting simulations with appropriately chosen stimuli on the other hand allows to quickly detect nonequivalence even in cases where both circuits only differ slightly. Combining both ideas naturally leads to an equivalence checking flow as illustrated in Fig. 7. Here, a limited number of $r \ll 2^n$ simulation runs with stimuli chosen according to some generation scheme (e.g., random computational basis states) are started in parallel with the $G \rightarrow \mathbb{I} \leftarrow G'$ equivalence checking routine according to some strategy (e.g., a strategy tailored toward verifying compilation flow results). Should any of these simulations (with stimulus $|\varphi_i\rangle$) yield different outputs in both circuits (i.e., a fidelity $\mathcal{F}(G|\varphi_i\rangle, G'|\varphi_i\rangle) \neq 1$) or should the equivalence checking routine be able to determine a final result not resembling the identity, the nonequivalence of the circuits under consideration has been shown, and all other executions can be aborted. On the other hand, the equivalence checking routine either manages to establish whether both circuits are equivalent or, in case it times out, leaves an indication (although no proof) that the circuits are likely

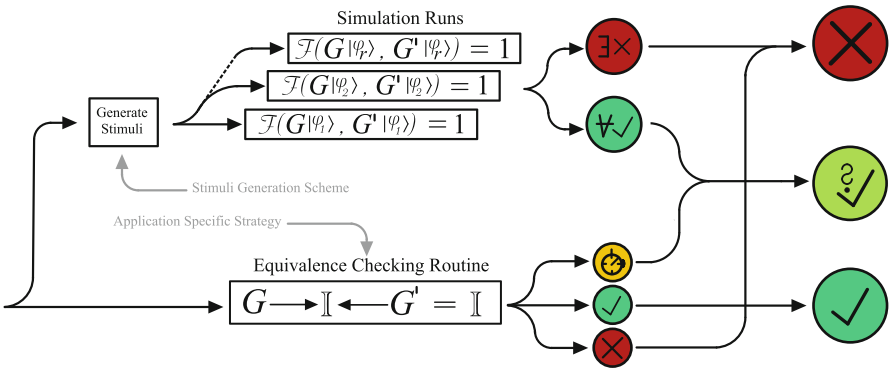


Fig. 7 Equivalence checking flow

to be equivalent, due to the fact that even small errors frequently affect the entire functionality.

The methodology described above is available as an open-source software package called *QCEC* (Burgholzer and Wille 2021b) which is part of the Munich Quantum Toolkit (MQT, formerly known as JKQ (Wille et al. 2020)) and available at github.com/cda-tum/qcec. It is mainly developed in C++, runs under any major operating system, and also provides Python bindings (including native integration with IBM Qiskit) in order to be as accessible as possible to its community.

Example 21. Assume a quantum circuit has been compiled using IBM Qiskit in the following fashion:

```
from qiskit import transpile

# create your quantum circuit
qc = <...>

# append measurements to save output mapping
qc.measure_all()

# compile to appropriate backend using some optimization
    level
qc_comp = transpile(qc, backend=<...>, optimization_level =<0 | 1 | 2 | 3>)
```

Then, after getting *QCEC* using the following: `pip install mqt.qcec`, verifying that the compiled circuit still realizes the originally intended functionality merely requires the following lines of Python:

```
from mqt import qcec

ecm = qcec.EquivalenceCheckingManager(qc, qc_comp)

ecm.run()

print(ecm.equivalence())
```

Conclusions

Considering the challenges currently seen in the verification of classical systems, it becomes increasingly important to prevent the emergence of a similar verification gap in quantum computing. This chapter lays the foundation for that by describing an advanced equivalence checking methodology for quantum circuits which takes the different paradigms of quantum computing not as burden but as an opportunity. Strategies for specific application scenarios can be easily derived which, e.g., allows to efficiently verify the results of compilation flows in many cases. Furthermore, inherent characteristics of quantum circuits allow for the conclusion that just a few simulations already provide an indication on whether two circuits are equivalent – even in the case of rather minor differences. In contrast to classical computing, this could eventually make verifying the results of sophisticated design flows feasible in general.

Acknowledgments This work received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement no. 101001318); was part of the Munich Quantum Valley, which is supported by the Bavarian state government with funds from the Hightech Agenda Bayern Plus; and has been supported by the BMK, BMDW, and the State of Upper Austria in the frame of the COMET program (managed by the FFG).

References

Amy M, Gheorghiu V (2019) Staq – a full-stack quantum processing toolkit. arXiv: 1912.06070

Amy M, Maslov D, Mosca M, Roetteler M (2013) A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits. IEEE Trans CAD Integr Circuits Syst 32(6):818–830

Barenco A et al (1995) Elementary gates for quantum computation. Phys Rev A, https://doi.org/10.1103/PhysRevA.52.3457

Bergeron J (2006) Writing testbenches using system verilog. Springer, New York

Biere A, Kunz W (2002) SAT and ATPG: boolean engines for formal hardware verification. In: International Conference on CAD, pp 782–785

Bookatz AD (2013) QMA-complete problems. arXiv: 1212.6312

Brandão FGSL, Harrow AW, Horodecki M (2016) Local random quantum circuits are approximate polynomial-designs. Commun Math Phys 346(2):397–434

Brennan J et al (2021) Tensor Network Circuit Simulation at Exascale. arXiv: 2110.09894

Burgholzer L, Wille R (2020a) Improved DD-based equivalence checking of quantum circuits. In: Asia and South Pacific Design Automation Conference

Burgholzer L, Wille R (2020b) The power of simulation for equivalence checking in quantum computing. In: Design Automation Conference

Burgholzer L, Wille R (2021a) Advanced equivalence checking for quantum circuits. IEEE Trans CAD Integr Circuits Syst, https://doi.org/10.1109/TCAD.2020.3032630

Burgholzer L, Wille R (2021b) QCEC: a JKQ tool for quantum circuit equivalence checking. Softw Impacts, https://doi.org/10.1016/j.simpa.2020.100051

Burgholzer L, Raymond R, Wille R (2020) Verifying results of the IBM Qiskit quantum circuit compilation flow. In: International Conference on Quantum Computing and Engineering

Burgholzer L, Raymond R, Sengupta I, Wille R (2021a) Efficient construction of functional representations for quantum algorithms. In: International Conference of Reversible Computation

Burgholzer L, Kueng R, Wille R (2021b) Random stimuli generation for the verification of quantum circuits. In: Asia and South Pacific Design Automation Conference
Burgholzer L, Bauer H, Wille R (2021c) Hybrid Schrödinger-Feynman simulation of quantum circuits with decision diagrams. In: International Conference on Quantum Computing and Engineering
Burgholzer L, Schneider S, Wille R (2022a) Limiting the search space in optimal quantum circuit mapping. In: Asia and South Pacific Design Automation Conference
Burgholzer L, Ploier A, Wille R (2022b) Exploiting arbitrary paths for the simulation of quantum circuits with decision diagrams. In: Design, Automation and Test in Europe
Chin-Yung L, Shiou-An W, Sy-Yen K (2011) An extended XQDD representation for multiple-valued quantum logic. IEEE Trans Comput 1377–1389, https://doi.org/10.1109/TC.2011.114
Cross AW et al (2021) OpenQASM 3: a broader and deeper quantum assembly language. arXiv: 2104.14722 [quant-ph]
Doi J, Takahashi H, Raymond R, Imamichi T, Horii H (2019) Quantum computing simulator on a heterogenous HPC system. In: International Conference on Computing Frontiers, pp 85–93
Drechsler R (2004) Advanced formal verification. Springer, https://doi.org/10.1007/b105236
Giles B, Selinger P (2013) Exact synthesis of multiqubit Clifford+T circuits. Phys Rev A 87(3):032332
Gottesman D (1997) Stabilizer codes and quantum error correction. Caltech
Green AS, Lumsdaine PL, Ross NJ, Selinger P, Valiron B (2013) Quipper: a scalable quantum programming language. SIGPLAN Not 48(6):333. arXiv: 1304.3390
Grover LK (1996) A fast quantum mechanical algorithm for database search. In: Proceedings of the ACM, pp 212–219
Guerreschi GG, Hogaboam J, Baruffa F, Sawaya NPD (2020) Intel Quantum Simulator: a cloud-ready high-performance simulator of quantum circuits. Quantum Sci Technol, https://doi.org/10.1088/2058-9565/ab8505
Hietala K, Rand R, Hung S-H, Wu X, Hicks M (2019) A verified optimizer for quantum circuits. arXiv: 1912.02250
Hunter-Jones N (2019) Unitary designs from statistical mechanics in random quantum circuits. arXiv: 1905.12053
Itoko T, Raymond R, Imamichi T, Matsuo A (2020) Optimization of quantum circuit mapping using gate transformation and commutation. Integration 70:43–50
Janzing D, Wocjan P, Beth T (2005) "Non-identity check" is QMA-complete. Int J Quantum Inform 03(03):463–473
Jones T, Brown A, Bush I, Benjamin SC (2018) QuEST and high performance simulation of quantum computers. Sci Rep, https://doi.org/10.1038/s41598-019-47174-9
Khatri S, LaRose R, Poremba A, Cincio L, Sornborger AT, Coles PJ (2019) Quantum-assisted quantum compiling. Quantum 3:140
Kitchen N, Kuehlmann A (2007) Stimulus generation for constrained random simulation. In: International Conference on CAD, pp 258–265
Klappenecker A, Rotteler M (2005) Mutually unbiased bases are complex projective 2-designs. In: International Symposium on Information Theory, pp 1740–1744
Kueng R, Gross D (2015) Qubit stabilizer states are complex projective 3-designs. arXiv: 1510.02767
Laeufer K, Koenig J, Kim D, Bachrach J, Sen K (2018) RFUZZ: coverage-directed fuzz testing of RTL on FPGAs. In: International Conference on CAD
Le HM, Große D, Bruns N, Drechsler R (2019) Detection of hardware trojans in SystemC HLS designs via coverage-guided fuzzing. In: Design, Automation and Test in Europe
Li G, Ding Y, Xie Y (2019) Tackling the qubit mapping problem for NISQ-era quantum devices. In: International Conference on Architectural Support for Programming Languages and Operating Systems
Maslov D (2016) On the advantages of using relative phase Toffolis with an application to multiple control Toffoli optimization. Phys Rev A 93(2):022311
Matsuo A, Hattori W, Yamashita S (2019) Reducing the overhead of mapping quantum circuits to IBM Q system. In: IEEE International Symposium on Circuits and Systems

Murali P, Baker JM, Javadi-Abhari A, Chong FT, Martonosi M (2019) Noise-adaptive compiler mappings for Noisy Intermediate-Scale Quantum computers. In: International Conference on Architectural Support for Programming Languages and Operating Systems

Nam Y, Ross NJ, Su Y, Childs AM, Maslov D (2018) Automated optimization of large quantum circuits with continuous parameters. npj Quantum Inf, https://doi.org/10.1038/s41534-018-0072-4

Nielsen MA, Chuang IL (2010) Quantum computation and quantum information. Cambridge University Press, Cambridge

Niemann P, Wille R, Miller DM, Thornton MA, Drechsler R (2016) QMDDs: efficient quantum function representation and manipulation. IEEE Trans CAD Integr Circuits Systems

Pednault E, Gunnels JA, Nannicini G, Horesh L, Wisnieff R (2019) Leveraging secondary storage to simulate deep 54-qubit Sycamore circuits. arXiv: 1910.09534

Schwinger J (1960) Unitary operator bases. In: Proc Natl Acad Sci 46(4):570–579

Seddon JR, Regula B, Pashayan H, Ouyang Y, Campbell ET (2020) Quantifying quantum speedups: improved classical simulation from tighter magic monotones. arXiv: 2002.06181

Siraichi MY, dos Santos VF, Collange S, Pereira FMQ (2018) Qubit allocation. In: International Symposium on Code Generation and Optimization

Smith KN, Thornton MA (2019) A quantum computational compiler and design tool for technology-specific targets. In: International Symposium on Computer Architecture, pp 579–588

Svore KM et al (2018) Q#: enabling scalable quantum computing and development with a high-level domain-specific language. In: Proceedings of RWDSL. arXiv:1803.00652

Viamontes GF, Markov IL, Hayes JP (2004) High-performance QuIDD-Based simulation of quantum circuits. In: Design, Automation and Test in Europe

Vidal G, Dawson CM (2004) Universal quantum circuit for two-qubit transformations with three controlled-NOT gates. Phys Rev A 69(1):010301

Villalonga B et al (2019) A flexible high-performance simulator for verifying and benchmarking quantum circuits implemented on real hardware. Npj Quantum Inf, https://doi.org/10.1038/s41534-019-0196-1

Vincent T et al (2021) Jet: fast quantum circuit simulations with parallel task-based tensor-network contraction. arXiv: 2107.09793 [quant-ph]

Watrous J (2018) The theory of quantum information. Cambridge University Press, Cambridge, 590pp

Wille R, Große D, Haedicke F, Drechsler R (2009) SMT-based stimuli generation in the SystemC Verification library. In: Forum on Specification and Design Languages

Wille R, Burgholzer L, Zulehner A (2019) Mapping quantum circuits to IBM QX architectures using the minimal number of SWAP and H operations. In: Design Automation Conference

Wille R, Hillmich S, Burgholzer L (2020) JKQ: JKU tools for quantum computing. In: International Conference on CAD

Wille R, Burgholzer L, Artner M (2021) Visualizing decision diagrams for quantum computing. In: Design, Automation and Test in Europe

Yuan J, Pixley C, Aziz A (2006) Constraint-based verification. Springer

Zulehner A, Wille R (2018) One-pass design of reversible circuits: combining embedding and synthesis for reversible logic. IEEE Trans CAD Integr Circuits Syst 37(5):996–1008

Zulehner A, Wille R (2019a) Compiling SU(4) quantum circuits to IBM QX architectures. In: Asia and South Pacific Design Automation Conference, Tokyo, pp 185–190

Zulehner A, Wille R (2019b) Advanced simulation of quantum computations. IEEE Trans CAD Integr Circuits Syst, https://doi.org/10.1109/TCAD.2018.2834427

Zulehner A, Wille R (2019c) Matrix-Vector vs. Matrix-Matrix multiplication: potential in DD-based simulation of quantum computations. In: Design, Automation and Test in Europe

Zulehner A, Paler A, Wille R (2019d) An efficient methodology for mapping quantum circuits to the IBM QX architectures. IEEE Trans CAD Integr Circuits Syst, https://doi.org/10.1109/TCAD.2018.2846658

Zulehner A, Hillmich S, Wille R (2019e) How to efficiently handle complex values? Implementing decision diagrams for quantum computing. In: International Conference on CAD

Index

A
AARCH64, 66, 68–70
Absolute error, 386
Abstract synthesis tree (AST), 1033
Abstractions, 48
Accelerator, 82, 208–209, 220, 226, 1037–1052
  design languages, 996
  model, 846–847, 1085–1086
  rich architecture, 844, 846
Access control policy, 1408
Accessibility, 953
Accuracy, 387
Accuracy optimized component verification, 1264–1265
  proving commutativity, 1267
  proving faithful rounding, 1265
  proving monotonicity, 1266
ACL2 preliminaries
  execution features, 1329–1332
  extension principles, 1327–1328
  intended domains and guards, 1330
  logic basics, 1325–1327
  must be equal, 1331
  single-threaded objects, 1332
  theorem prover, 1328–1329
Activation motion compensation (AMC), 365
Active mask, 544
Active pixel sensor (APS) array, 93
Acyclic graph, 1033
Adaptive systems, 522–524
Adder, 396–397, 429, 430, 447, 449, 1031
Address event representation (AER) protocol, 334
Address generation, 1156–1157
Address generation unit (AGU), 1165
Address interleaving, 613, 618
Address space layout randomization (ASLR), 187
Advanced configuration and power interface (ACPI) standard, 581
Advanced datapath verification
  accuracy challenges, 1261
  managing inconclusive proofs, 1259
Advanced encryption standard (AES), 238
Advanced matrix extensions, 59
Advanced RISC machines (ARM), 64
Advanced vector extensions (AVX), 76
Advanced video coding (AVC), 216, 218, 219, 221–223
Aerosols, 704
Affine dataflow (ADF), 1119
Aggregate performance metrics, 633
Aggressive scheduling, 1181, 1182
Aging/stress-induced faults, 283–284
Air monitoring, 704
Algorithms, 880
Alias analysis, 853, 861, 1156
Alliance for Open Media (AOMedia), 217
Allocation, 849, 851, 855, 862, 865–867
ALWANN methodology, 1047–1048
AMBER system, 228
AMD/Xilinx, 147, 510–512, 515, 517, 518
Amdahl law, 50, 52, 601–603
AMX system architecture, 77
Analog circuits, 335–338
Analog systems, 1358
Analog-to-digital converter (ADC), 652, 659, 667, 668, 674–682
Analytical reliability estimation, 288–289
Ancillary qubits, 1428
ANGR, 1370, 1374
And-inverter graphs (AIGs), 1033, 1284, 1287, 1293, 1295, 1300, 1317
Anti-dependency, 40
Application exploration, 937–940
Application-level mapping
  CPU and CGRA, partitioning between, 496
  SDF, 497–498
Application model, 921

© Springer Nature Singapore Pte Ltd. 2025

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-97-9314-3

Application programming interfaces (APIs), 402, 403, 407, 1109
Application scenarios, 933
Application specific instruction set processors (ASIP), 260, 418–420, 423, 451, 458, 459, 466, 470, 496, 809, 812–814, 817–819, 821, 824, 827, 828, 830–833, 843, 848, 850, 861, 864, 868, 1151, 1161, 1162, 1164–1184
Application specific integrated circuits (ASICs), 257, 260
Approximate arithmetic components, 1030
  arithmetic error metrics, 1035–1036
  automated approximation methods, 1032–1034
  design methodologies, for approximate components, 1030–1034
  error metrics and evaluation analysis, 1034–1037
  general error metrics, 1036
  manual approximation methods, 1031–1032
  quality evaluation, 1036–1037
Approximate carry select adders, 1032
Approximate circuits, 1033, 1035–1037, 1039
Approximate computing, 1028–1030
  approximate arithmetic components, 1030–1037
  cross-layer approximations, for error-tolerant applications, 1052–1055
  energy and performance efficiency, of DNN inference, 1055–1062
  hardware accelerators, error-tolerant applications, 1037–1052
Approximate full adders, 1032
ARC APEX, 820–821
ARChitect, 820
Architectural description language (ADL), 877
Architectural Last Branch Records (LBRs), 60
Architecturally correct execution (ACE), 288
Architectural registers, 41
Architectural vulnerability factor (AVF), 288–289
Architecture analysis, 956
  HW/SW performance optimization and validation, 959–960
  macro-architecture specification, 956–958
Architecture description languages (ADLs), 810, 950, 1151, 1168
  ANDES ACE, 821–822
  applications of ADL-based design, 831–833
  ARC APEX, 820–821
  automatic synthesis of custom instructions for an application, 824
  capabilities, 831–833
  characteristics of, 814–816
  classical era (1990–2000), 813
  Codasip CodAL, 821
  exploration, synthesis and validation of programmable architectures, 811
  EXPRESSION, 817
  first industrial era (2000–2010), 813
  generation of hardware implementation, 827–828
  generation of software tools, 823–824
  instruction-set simulator generation, 825–827
  LISA, 818–819
  MIMOLA, 816–817
  nML, 817–818
  PEAS, 819
  RISC-V CHISEL, 822–823
  second industrial era (2010–2020), 814
  specification and modeling capabilities, 812
  specification-driven, simulation-based, verification, 830–831
  Tensilica TIE, 819–820
  top-down verification, 828–831
  types of, 814
  validation, 829–830
Architecture exploration, 953, 956, 973, 983
Architecture-level Laws and Models, 51
Architecture model, 921
Areal power density, 634
Arithmetic and logic units (ALU), 174
Arithmetic circuits
  direct verification, 1304–1305
  floating-point addition, 1307–1308
  floating-point division and square root, 1311–1313
  floating-point multiplication and fused multiply-add, 1310–1311
  floating-point operations, 1306–1307
  integer multiplication, 1308–1310
Arithmetic error metrics, 1035–1036
Arithmetic format support, 408
Arithmetic logic units (ALU), 6, 535
Arithmetic research, 382
ARM7-32 bits, 66
ARM family of RISC processors, 62
ARM Trustzone, 173
Array multiplier, 1032
Artificial dependencies, 855
Artificial intelligence (AI), 90, 322, 586, 648

Artificial neural network (ANN), 323, 325, 349, 350, 354, 372
accuracy, 353–354
algorithm-driver architecture optimization, 365
application level design, 355
architecture design, 355
bit-width reconfiguration, 363–364
computation reuse, 364–365
design abstractions and trade-offs, 355–357
design space exploration, 367
ecosystem, 354
frameworks and libraries, 356
functionality, 351
hardware architecture design, 356–357
hardware software co-design, 356
low latency inference, 365–366
miscellaneous networks, architectures for, 367–368
open-source designs, 369–370
optimization, 355
performance, 351–352
power consumption, 352–353
reliability and security, 366–367
selective ANN architectures and circuits, 357–370
silicon area, 353
3D memory, computation in, 363
training architectures, 368–369
types of dataflow, 361–363
Assembly queues (AQ), 1102
Assertion-based property, 1407
Assume guarantee, 1256
Asynchronous circuit, 339, 341
Asynchronous design methodology, 341
Atomic memory operations (AMO), 72
Audio compression, 885
AutoAx methodology
library pre-processing, 1040
model construction, 1040–1041
model-based design space exploration, 1041–1042
AutoDNNchip, 367
Automata-theoretic construction, 1207
Automated approximation methods, 1032–1034
Automated Behavioral Approximate CircUit Synthesis (ABACUS), 1033
Automated capabilities, 833
Automated verification, 1192, 1193
Automatic methodology for Sequential Logic ApproximatioN (ASLAN), 1033
Autonomous mode management (AMM), 94, 120
Auto-vectorization, 1158, 1173
Average memory access time (AMAT), 621, 622
Average temperature reduction, 576
Average-case arithmetic error, 1035
AVX-512 Conflict Detection Instructions, 77
AVX-512 extension, 76
AVX-512 Foundation, 77
AVX-512 Prefetch Instructions, 77

B

Backward error recovery (BErR), 297
Balanced processor architectures, 205
Bandwidth, 895
Bank conflicts, 618, 620
Bank-level parallelism (BLP), 620
Barrett's reduction, 253
Base register, 8
Basic block, 1154
Basic linear algebra subprograms (BLAS), 407, 410
Basic logic element (BLE), 425
Basic RISC-V model, 71
Batch normalization, 938
Batch systems, 128
Bathtub curve, 279
Benes distribution network, 108
Berkeley Design Technology, 892
Bernoulli sampling, 249, 250, 262
bfloat16 (BF16) format, 395
Bias temperature instability (BTI), 284, 306
Bidirectional skipping mechanism, 107
Big data workloads, 536
Binarized convolutional neural networks, 1013
algorithm, 1013–1014
data vectorization, 1016–1017
evaluation, 1017
HeteroCL, 1017–1019
line buffers and window buffers, 1016
pipelining and unrolling, 1014–1016
Binary code analysis, 1335–1336
Binary convolutional neural network (BCNN), 361
Binary decision diagrams (BDDs), 1195, 1209, 1215, 1218, 1250, 1284, 1287, 1293–1295, 1299, 1304, 1305, 1307–1309, 1317, 1402
Binary execution phase, 1373
Binary half-adders, 396
Binary-level in-vivo analysis, 1371
Binary multiply-accumulate (BMAC) operations, 1013
Binary translation, 50, 923

Binary two-input Boolean AND operation, 1393–1394
Binding, 849, 857, 858
Binomial sampling, 262
BitBlaze, 1370
Bit error rate (BER), 580
Bitlet Model, 52
Bitline, 440
Bit-level model checking, 1195
  BDDs, 1209
  computing reachable states, 1211
  design simplification techniques, 1235–1239
  linear time temporal logic, 1207
  liveness properties, algorithms for, 1229–1235
  notation, 1210
  safety properties, algorithms for, 1212–1229
  SAT, 1209
  simple counter, 1205–1207
  symbolic successor computation, 1210
  transition system, 1208–1209
BITMAN, 515
Bitstreams, 510
Bit-width reconfiguration, 363–364
Blackboxed modules, 1251
Blackboxing, 1254
Blackbox model, 926
Bloch sphere, 1433
Block RAM (BRAM), 419, 421, 440, 442–447, 451, 454, 460
Bluetooth, 158
BMF-based Logic Approximate SYnthesiS (BLASYS), 1034
Boolean circuits, 1196
Boolean domain, 1196
Boolean function, 1035, 1196
Boolean matrix factorization (BMF), 1034
Boolean operators, 1275
Boolean satisfiability (SAT) solvers, 483, 1209
Boolean variables, 1208
Bounded liveness checking
  counter based translation, 1232
  kLiveness, 1232–1233
  stabilizing constraints, 1233–1234
Bounded model checking (BMC), 1215, 1216, 1218, 1223, 1273, 1289
Brain floating point, 1012
BrainScales, 345
Branch History Register (BHR), 32
Branch penalty, 30
Branch Prediction Unit (BPU), 183
Branch predictor, 29
Branch resolution, 27
Branch speculation, 29
Branch Target Buffer (BTB), 30, 34, 183
BTB hit, 30
Bug hunting, 1256
Built-in Self Recovery (BISR), 297
Built-in Self Test (BIST), 297
Bulk-Synchronous Parallel (BSP) model, 52
Bus-cycle accurate simulation, 924
Bus lock, 60
Bus/memory optimization, 958, 960
Bypass network, 23

C
Ca^{2+}-induced Ca^{2+} release (CICR), 311
Cache block granularity, 623
Cache block interleaving, 618
Cache coherence, 614, 622–624, 1342
  coherence protocol, 625–626
  exchanging coherence messages between cores, 624–625
  need for coherency with write-back caches, 623–624
Cache optimization, 959
Cache partitioning, 622
Cache slicing, 611
Caches, 409, 1405–1408
Caching, 895
Cadence Genus Synthesis, 1060
Caffe, 995
Cambricon series, 358–359
Capability Hardware Enhanced RISC Instructions (CHERI) ISA, 1337
Capacity wall, 866
Carry chain, 430, 447
Carry-lookahead adder (CLA), 397, 1031
Cascade-2D designs, 764, 765
Central processing unit (CPU), 5, 128, 132, 145, 202–204, 206, 323, 887, 909
  memory parallelism, 206–208
  multi-core CPUs, 134–137
  pinning, 629
  single-core CPUs, 132–134
  types, 204, 205
Ceva processors, 909
CGS-16 architecture, 793
CGS-64 architectures, 799
Chaining, 856, 863
CHiMPS, 994
CHiMPS target language (CTL), 994
Chip multiprocessors, *see* Multicore CPUs
Chiplet-based multicore design, 638–639
Chooser, 33

ChordMap, 497
Chosen-ciphertext attack (CPA), 246
Chosen-plaintext attack (CCA), 246
Chromosome, 928
CIFAR-10, 939, 940
CIM crossbar array, 651
Circuit-level design considerations, 335–342
Circuit properties, 1281
  mathematical model of, 1282–1283
Circuit simulation
  mathematical model of, 1279–1281
  and undefined values, 1276–1278
Cirq, 737
Clang, 1150, 1175
CLI stack project, 1337
Click-based link-joint asynchronous circuit, 339
Clique covering, 858
Classical circuits, 1420–1421
Classical computing, 1415
Classical simulation, 1418
Clock-gating method, 570
Clock sector, 438, 439
Clock skew, 437–439
Clock tree synthesis (CTS), 438, 767
Closest vector problem (CVP), 244
Cloud Columba, 701
Cloud systems, 146–147
CMOS technology node, 94
Coarse-grained dataflow, 1003
Coarse-grained reconfigurable array (CGRA), 360, 402, 405–406, 410–412, 466, 467, 470, 500, 846
  application-level mapping, 496–498
  basic CGRA architecture, 471
  complex scheduling mechanisms, 412
  CPU, 474–475
  graph theory inspired techniques, 484–491
  handling loops with control flow, 499
  heuristic approaches, 478–483
  high degree of hardware–software codesign, 412
  historical context, 468–471
  homogeneous and heterogeneous CGRA, 471
  lack of standardisation, 412
  mathematical optimization techniques, 483
  memory address generation, 494
  memory aware compilation, 493–494
  memory hierarchy, 473
  modulo scheduling, 475–476
  MRRG, 476–477
  nested loop mapping, 494–496
  on-chip network, 472–473
  poor programmability, 412
  scalable CGRA mapping, 499–500
  spatial CGRA, 472
Coarse-grain sparsification, 788
COD, 1376, 1377, 1379
Codasip CodAL, 821
Code-based cryptography, 242
Code, Data, and Stack segment, 56
Coded tree blocks (CTBs), 219
Code selection, 1160, 1176–1177
Coding tree units (CTUs), 220, 222, 227–229
Cognitive radio systems, 522
Coherence state, 623
Collaborative Macro-Function Units (CMFUs), 297
Collision avoidance system, 130
Columba, 700, 701
Combinational equivalence, 1245–1246
Combinational redundancy removal, 1236
Commercial off-the-shelf (COTS) Linux kernel modules, 303, 1375–1379
Commit, 40
Common sub-expression elimination, 1159
Common Verification Environment (CVE), 1315
Communicating hardware processes (CHP), 341
Communicating sequential processes (CSP), 341
Commutativity, 1267
Comparators, 23
Compatibility graph, 490, 858
Compilation phases and dependencies, 1154–1155
Compilers, 823
  back-end, 1159–1160
  construction, 1149
  framework, 1149, 1151
  front-end, 1155
  middle-end, 1156–1159
  optimization, 851
  techniques, 412
Complementary metal-oxide-semiconductor (CMOS) technology, 564, 565, 648, 649
Complete ADLs, 828
Completion axioms, 1330
Complex instruction set computer (CISC), 36, 49, 54, 56, 57, 59, 60, 809
Complex programmable logic device (CPLD), 424
Complexity analysis, 1299–1300

Complexity management complexity analysis, 1299–1300
simulation complexity, 1297–1298
weakening, 1300–1301
Compositional timing, 971
Compressed sparse column (CSC), 103, 105, 106
Compressed sparse row (CSR), 103
Compression ratio, 104
Compute-in-memory (CIM)
advantages, 649
circuit techniques, 672–680
designs, 659
frameworks, 681
hardware implementations, 667–670
hierarchical architecture, 652–653
network mapping strategies, 652–653
non-idealities, 671
pipeline design, 660–664
principle, 650–652
quantization techniques, 664–667
Computer-aided design (CAD), 418, 420, 421, 428, 429, 431–433, 435, 437, 438, 444, 446, 689, 691, 1244
Computer aided verification, 1199, 1200
Computer architecture, 322
Computer arithmetic and arithmetic architectures
absolute error, 386
8087 coprocessor chip, 385
fixed-point arithmetic, 389–390
floating-point arithmetic, 390–396
8087 floating-point coprocessor, 384
FLOPS, 387
hardware implementations, 396–398
integer arithmetic, 387–389
machine epsilon, 387
numerical precision, 387
positional notation, 386
radix, 385
relative error, 386
timeline of events, 383
ULP, 387
Computer hardware, 1194
Computer-unified device architecture (CUDA), 408
Computing resources, 521
Computing systems, 128
Concolic execution, 1367, 1369
Concolic testing, 1369–1370
COTS Linux kernel modules, 1375–1379
hardware/software co-validation of systems-on-chips, 1380–1385
Concrete execution, 1370
Concurrent error detection (CED), 268
Conditional control flow, 8
Conditional direct branches, 6
Cone of influence (COI), 1298, 1400–1401
Configurability, 906
Configuration controller, 459
Configuration SRAM, 418, 424, 442, 452, 459
Conjunctive normal form (CNF), 1036, 1209
Connection block multiplexers, 425, 431, 442
Connection Operation Graph (COG), 817
Connectivity, 202
Constant failures, 280
Constant folding, 1159
Constant memory, 546
Content-addressable memories (CAMs), 613
Context switch, 129, 600, 629
Control and data-flow graph (CDFG), 1154
Control area network (CAN), 157
Control-channel routing, 693
Control dataflow graph (CDFG), 1011
Control dependencies, 25
Control-flow graph, 1154
Control-flow optimization, 1158
Control hazards, 18, 26–35
Controlled NOT (CNOT) gate, 725
Controller, 847, 859, 863, 867
Control Register (CTRL), 1383
Control unit, 11
Conventional image sensor systems, 95
Conventional loop pipelining, 1002
Conventional target data rate control, 98–99
Convergence bound, 1312
Convolutional neural network (CNN), 73, 303, 349, 538, 552, 649–651, 654, 656, 657, 661, 937–939
CoPR, 515
Counterexample guided abstraction refinement (CEGAR), 1238, 1239
Counterexample to induction (CTI), 1226
Coyote, 519
CPI, 36
Craig interpolation, 1221
CRETE, 1371–1375
Cross-layer approximations
cross-layer methodology, for optimizing DNNs, 1054–1055
hardware and software-level approximations, 1052–1053
Cross-layer design approach, 303
Cross-layer reliability (CLR), 301–304
Cross-layers optimizations, 79, 80, 82–84
Crossover, 928
Crosstalk, 189, 437, 439, 729
Cryptosystems, 284

C-to-RTL equivalence checking, 1195–1197
CUBLAS, 410
CUDA code, 82, 84, 147, 538
CUDA Pitfall Detector for Real-Time Systems (CUPiD^{RT}), 143
Cumulative distribution table (CDT) sampler, 250
Current state variables, 1208
Custom architectures, 407
Custom instruction set extension, 846
Custom instructions, 901
Custom integrated circuits, 407
Customized memory hierarchy, 998
Custom precision floating-point data types, 1012
CUTE, 1370
Cyber-physical systems (CPS), 130, 707, 1028
Cycle accurate simulation, 908, 923
Cycle-based ISS, 974
Cycle-level simulation
  configurability, 906
  definition, 905
  metrics and system partitioning, 905
  open-source simulators, 906
  optimisation, 906
  performance analysis, 905
Cycle-specific weakening, 1300, 1301
Cycles per instruction (CPI), 36, 927
Cyclic redundancy check (CRC), 459
Cyclic shift registers, 1008
Cyclo-static dataflow (CSDF), 1119

D
Dark silicon, 563, 842
DART, 1370
Data bounce, 187
Datacenter FPGA, 456
Data center network (DCN), 584, 585
Data communication, 573
Data dependencies, 895
Data Encryption Standard (DES), 238
Data-flow analysis, 1156
Dataflow engines (DFEs), 994
Data-flow execution model, 1091
Data flow graph (DFG), 36, 473, 476, 479, 486, 488, 492, 1154, 1250
Dataflow process network (DPN), 1121
Dataflow processor (DPU), 369
Data forwarding, 571
Data hazards, 18–25
Data-level parallelism, 536, 1163
Data parallelism, 1071, 1080
Datapath, 846, 847, 856, 863, 1257–1259, 1299, 1302, 1304, 1306, 1310, 1314
Data rate control, 101
Data store, 611
Data type customization, 1011, 1164
  automatic bitwidth optimization, 1011
  custom precision floating-point data types, 1012
  float to fixed-point conversion, 1012
Data types and operations, 879, 880
Data vectorization, 1010, 1016–1017
DDR5, 211, 613, 620
Dead-code elimination, 1159
Deadline Monotonic (DM) scheduler, 133
Debug, 952, 1254
Decision tree classification (DTC), 371
Decision variables, 917
Decode, 6
Decoherence, 728
Decoupled access-execute (DAE), 494, 1008
Deep learning (DL), 233, 309, 423, 430, 460, 649, 650, 681
Deep learning accelerator (DLA), 956, 959
Deep neural network (DNN), 648–650, 652, 654, 659, 660, 664, 665, 671, 680, 682, 753, 797, 1046–1047
  ALWANN methodology, 1047–1048
  architecture description, 789
  cross-layer methodology, 1054–1055
  energy and performance efficiency, of DNN inference, 1055–1062
  energy-efficiency of, 791–793
  evaluation and experiments, 1049–1051
  speech recognition, 786
  training and classification, 787
Defaxiom principle, 1327
Defchoose principle, 1327
Definitional principle, 1327, 1328
Degree of fault tolerance, 313
Delft Workbench (DWB), 994
Delite hardware definition language (DHDL), 996
Demand bound function, 134
Dennard scaling, 596
Dennard's Power Scaling, 51
Denormal numbers, 391
Dense linear algebra (DLA), 403
Design exercise, 1257–1259
Design rule checking (DRC), 754
Design space exploration (DSE), 302, 350, 367, 568, 586, 850, 866, 867, 916–919, 940, 941
  analytical fitness evaluation, 926–927
  application exploration, 937–940

Design space exploration (DSE) (*cont.*)
design/compile-time DSE, 567
design space, searching, 927–932
GA-based DSE, 928–932
hybrid DSE, 568
multi-application workload models, 932–937
multiple design criteria, 919
run-time DSE, 567
simulative fitness evaluation, 922–925
single design point, evaluation of, 922–927
taxonomy, 920
Y-chart based DSE, 920–922
design space exploration (DSE), 563
design-time profiling, 584
Design under verification (DUV), 1420
Destination operand, 7
Destination register (dreg), 7
Detection-based countermeasures, 194
Deterministic circuits, 1281
DFGNet, 482
Dhall effect, 134
Dhrystone, 888
DianNao series, 357–358
Differential IO, 436
Digital circuits, 338
Digital circuit simulation, 1270
Digital-corporation, 85
Digital image processing techniques, 649
Digital microfluidic biochips (DMFB), 701, 709
air monitoring, 704
droplet routing, 706–707
point-of-care diagnostics, 703
scheduling and module placement, 705–706
single-cell isolation and analysis, 703
synthesis methods, 704–707
technology platforms and applications, 701–704
Digital signal processing (DSP), 243, 253, 303, 447–453, 890, 1151, 1175
Digital signature schemes (DSS), 243
Digital-to-analog converters (DACs), 673, 674, 676
Digital twin, 983
Direct conditional branches, 26
Direct drive switch, 434
Directed acyclic graph (DAG), 144, 145, 1176
Directed graph, 484
Direct hardware execution, 422
Direct-memory access (DMA), 471, 474
Direct Memory Interface (DMI), 969
Directory-based cache coherence, 624
Direct unconditional jump, 26
Direct verification, 1304–1305
Discrete cosine transform (DCT), 216
Discrete Fourier transform (DFT), 251
Discrete Gaussian sampling, 249–250, 261–263
Discrete Ziggurat sampling, 250, 262
Dispatch, 40
Displacement, 8
Distributed channel-storage architecture, 694
Distributed memory models, 1073
Distributed shared memory models, 1073
Dividers, 398
DNN architecture, 800
DNN topology, 786
DNN workloads, 800
Domain-specific fault-tolerance, 303
signal processing, 303
wireless communication, 304
Domain-specific language (DSL), 995–996
Double-bit-error-correcting (DEC), 298
Double data rate memory (DDR), 436, 455, 456
Double-nibble-error-detecting (DND), 298
DRESC compiler, 478
Droplet dragging, 708
Droplet holding, 708
Droplet routing, 706–707, 715
Droplet sensing, 708
DSP mapping, 1000
Dual inline-memory module (DIMM), 299
Dual modular redundancy (DMR), 291, 556
Dynamically allocated, multi-queue buffer (DAMQ), 493
Dynamic branch predictors, 30
Dynamic dataflow, MoCs
dataflow process networks, 1121
Kahn process network, 1120, 1121
πSDF, 1122, 1124
reconfigurable dataflow, 1122
Dynamic dissipation, 565
Dynamic fixed point quantization, 665
Dynamic multi-issue, 1163
Dynamic out-of-order, 39
Dynamic partial reconfiguration (DPR), 297
Dynamic power management (DPM), 226, 570
Dynamic-priority scheduling, 133
Dynamic rail analysis, 781
Dynamic random access memory (DRAM), 203, 204, 207, 208, 211, 298, 299, 304, 569, 613, 614, 616, 618–620, 648, 667, 681
Dynamic techniques, 1194
Dynamic thermal management (DTM), 229–232

Dynamic voltage and frequency scaling (DVFS), 352, 569, 572, 574, 576–580, 582–584, 586, 608, 609, 630, 1028
Dynamic weakening, 1256, 1301
DyRACT, 520

E
Edge-centric modulo scheduling (EMS), 478–480
Effective address, 8
Elastic circuit, 862
Elasticity, 1099
Electrical masking, 285
Electromigration (EM), 284
Electronic control unit (ECU), 961
Electronic design automation (EDA), 689, 753, 809, 1192
Electrowetting on dielectric (EWOD), 701, 702, 708
Embedded multi-die interconnect bridges (EMIB), 459
Embedded SRAM (eSRAM), 446
Embedded systems, 809, 810, 813, 823, 824
Embench, 890
Emerging non-volatile memories (eNVMs), 649, 667, 669, 671, 672, 681, 683
Empirical model, 926, 927
Emulation testing, 1194
Encapsulation principle, 1327
Encoded instruction-word, 1163
Endianness, 815
Energy and performance efficiency
  quantization, 1057–1058
  self-healing and non-self-healing designs, on DNN accuracy, 1058–1062
  structured pruning, 1055–1057
Energy consumption, 572, 573
Energy efficiency, 422, 552, 635
  revisiting compute cores and pipeline, 553–554
  revisiting register file, 554–555
Energy harvesting, in IoT edges, 92
  power converter efficiency, effects of, 95
  processing pipeline, effects of, 94
  self-powered image sensor system, with autonomous mode management, 93–94
  SRAM leakage energy, 95
  unit pixel size, effects of, 94–95
Energy minimization, 572
  communication, 573
  computation, 572–573
  memory, 574
Energy scavenging, 92
Enhanced hardware feedback interface (EHFI), 60
ENQCMD/ENQCMDS instructions and virtualization support, 59
EPIMap, 488
Equivalence-based verification, 1272
Equivalence checking, 1195–1197, 1436–1437
Error injection, 310
Error masking, 286, 290
Error probability, 292, 1036
Error-tolerant applications
  cross-layer approximations, 1052–1055
  hardware accelerators, 1037–1052
Essential semantics, 849
Estimation analysis, 893, 894
Euclidean space, 244
European Telecommunications Standards Institute (ETSI), 241
Evict+Time attacks, 179
Evolution of multicore CPUs
  chiplet-based multicore design, 638–639
  heterogeneous CPU cores, 637–638
  systems-on-chip, 635–636
Evolutionary algorithm, 481, 1034
Evolutionary Piecemeal Training (EPT), 938–940
Execution circuit, 7, 9
Execution units (EUs), 138
Exhaustive verification, 1192
Existing processor, 882, 883
Explicit and implicit flows, 1394–1395
EXPRESSION, 817
Expression optimization, 1159
Extending configurable processor, 883
Extensibility, 1371
Extension principles, 1327–1328
ExTensor, 106, 107
External faults, 282
Extreme waterfalling, 1260
Eyeriss series, 360

F
FAIR, 1234–1235
Fallout, 186
False dependency, 40
False sharing, 623
Fast Fourier transformation (FFT), 251
Fast-functional simulation, 905
Fast heuristics, 584
Fault analysis, 557
Fault attacks, 268–269

Fault injection (FI)
emulated FI, 288
physical FI, 288
simulated FI, 288
Fault masking, 284–285
Fault mechanisms
aging/stress-induced faults, 283–284
external faults, 282
Fault model, 281
Fault-tolerance
activities, 290–291
in AI/ML, 309–313
detection, 290
diagnosis, 290
domain-specific, 303–304
emerging memory technologies, 304–306
fault avoidance, 290
isolation, 290
NVMs, reliability issues in, 306–309
recovery, 291
redundancy, 291–294
repair, 290
self-repair, 310–313
Fault-tolerant computation, 294
multi-core computing, 296–297
reconfigurable computing, 297
single-core computing, 294–296
Fault-tolerant memory/storage, 298
cache/on-chip SRAM, 299
main memory/DRAM, 299
storage, 299–300
Fault-tolerant on-chip communication, 300–301
Faults
intermittent, 281
multiple bit-flip, 281
permanent, 281
single bit-flip, 281
stuck-at, 281
transient, 281
Feature selection (FS), 935
Feature-space maximum likelihood linear regression (fMLLR), 787
FENCE instruction, 72
Ferroelectric field-effect transistor (FeFET), 666, 667, 670, 681
Fetch, 6, 7
Fidelity, 1041
Field programmable gate array (FPGA), 148, 297, 402, 405, 410, 412, 467, 470, 480, 481, 508, 509, 519, 525, 526, 845, 847, 848, 850–852, 861, 862, 864, 865, 868, 953, 990, 992, 1019
adaptive systems, 522–524
applications, 422–423
binarized convolutional neural networks, 1013–1019
computing infrastructure and virtualization, 520–521
configuration, 459, 510–511
data type customization techniques, 1011–1012
design compilation, 521
designing partially reconfigurable systems, 512–517
DSP blocks, 447–453
dynamic partial reconfiguration, 519–525
hardware debugging, 410
high specification to deployment time, 410
HLS compilers and programming models, 992–996
interposers, 457–459
key compiler and synthesis optimizations, 997–1012
limited resources, 410
machine learning, 524
managing partial reconfiguration, 517–519
memory customization techniques, 1007–1011
methodology and tools, 420–422
network-on-chip, 455–457
on-chip memory, 439–447
parallelization techniques, 1004
pipelining techniques, 998–1004
processor subsystems, 453–455
programmable clock distribution networks, 437–439
programmable IO, 435–436
programmable logic blocks, 423–430
programmable routing, 430–435
reliability and harsh environments, 524–525
Fine-grained dataflow, 1003
Finite impulse response filter (FIR filter), 422
Finite sequence, 1279
Finite state machine (FSM), 847, 863, 1110, 1111, 1206, 1246
Finite-state transition system, 1208
Finite stimulus trace, 1280
Finite trace, 1280
Firmware, 569
DPM, 570
DVFS, 569
virtualization, 571
First-Come-First-Served (FCFS), 129
First-ready, first-come first-serve (FR-FCFS), 621, 622
Fitness function, 917, 919

Fixed-point arithmetic, 389–390
Fixed-priority scheduling, 132
Fixed virtual prototypes (FVP), 955
Flash-based configuration memory, 510
Flash translation layer (FTL), 978
Flexibility-Efficiency Trade-off, 53
FlexRay, 157
Flip-flop (FF), 425
Floating-point approximate circuits, 393
Floating-point arithmetic, 390
  circuits, 393
  IEEE 754, 390–393
  posit arithmetic, 394–395
Floating point operations (FP), 57, 1306–1307
Floating-point operations per second (FLOPS), 387
Floating Point Units (FPU), 174
Float-to-fixed conversion, 1012
Floorplanning, 512, 515
Flow-based microfluidic biochips (FBMBs)
  architectural synthesis framework, 692
  architecture design of control layer, 693
  architecture design of flow layer, 691–692
  control-channel routing, 693
  design automation, 693
  design tasks for, 691–693
  high-level synthesis, 692
  physical design, 692
  structure, 690
  synthesis methods for codesign of control and flow layers, 699–701
  synthesis methods for control layer, 696–699
  synthesis methods for flow layer, 693–696
  valve addressing, 693
Flow relation, 1392
Flushing correspondence, 1340
Flush phase, 178
Flush+Reload attacks, 178
Flynn's taxonomy, 52
Force-directed list scheduling (FDLS), 706
Foreshadow, 184
For-loop, 1158, 1182
Formal analysis, 1391
Formal equivalence
  combinational equivalence, 1244–1246
  sequential equivalence, 1246–1247
  transaction-based equivalence, 1247–1248
Formal equivalence verification (FEV), 1273
Formal language, 1325
Formal property verification (FPV), 1273
Formal specification languages, 810
Formal verification, 1194, 1271, 1272, 1322, 1404
Forward error recovery (FErR), 296
Forwarding network, 22–24
Fracturable LUT, 426–428, 430
Frequency- and time-domain analysis, 782
Frequency reconstruction error, 313
Full bitstream, 512, 517
Fully programmable valve array (FPVA), 690
Functional approximation, 1063
Functional fidelity, 969
Functional flows, 1406
Functional Mock-up Interface (FMI), 961
Functional reliability, 280, 286, 578–580
Functional-unit binding, 857
Functional unit (FU), 466, 470, 471, 1164
Function in-lining, 1158
Function Vulnerability Index (FVI), 289
Fused multiply-addition (FMA), 76, 1311

G

Gadget-based secret transmission, 188
Gate error, 728
Gate excitation functions, 1278
Gate leakage, 566
Gaussian cumulative distribution function, 250
Gaussian distribution, 97, 250
Gaussian image filter, 1045
Gaussian mixture model (GMM), 97, 786
General error metrics, 1036
General Matrix Multiply (GEMM), 467, 468
General purpose GPU (GPGPU), 532, 535, 538, 540, 550
General purpose graphics processing units (GPGPUs), 137, 408–410
Generalized matrix multiplication (GEMM), 101, 103
Generalized STE (GSTE), 1316
Generic interrupt controller (GIC), 979
GENESIS, 995
Genetic algorithm (GA), 928–932
Global history register (GHR), 33
Global 2-level branch predictor, 33
Global memory, 545
Global predictors, 32
Global quantum stimuli, 1435
Global scheduling, 136
Global two-level predictors, 32
GNU Compiler Collection (GCC), 1150, 1170
GoAhead, 515
Google TPU, 359–360
Gradual underflow, 391
Graph drawing based technique, 491
Graph epimorphism based technique, 488
Graph homeomorphism, 485, 486

Graphical user interface (GUI), 820
Graphics, 885
Graphics cards of machine learning accelerators, 82
Graphics processing unit (GPU), 132, 137, 138, 145, 146, 202, 233, 323, 408, 419, 532, 599, 636, 887, 990, 1085
  accelerated DNN computations, 789
  access-aware variable mapping to memory, 547–549
  advanced warp schedulers, hiding memory access latency with, 550–551
  arithmetic format support, 408
  caches, 409
  constant memory, 546
  energy efficiency, 552–555
  execution model, 536–538
  global memory, 545
  GPU for general purpose computing, 536–549
  graphics pipeline, 533–535
  hardware architecture, 138–140, 540–545
  initial GPUs, 408
  L1 and L2 caches, 547
  performance, 550–552
  programming interface, 538–540
  register file, 409, 542
  reliability, 555–557
  shader pipeline, 540–542
  shared memory, 546–547
  simplified architecture of traditional GPU architecture, 534
  SIMT stack, 544–545
  single GPU, scheduling tasks on, 140–143
  stream processing, 409
  texture memory, 546
  threading model, 139–140
  throttling memory access latency, 551–552
  vectorization, 408
  warp scheduler, 543–544
Graph isomorphism, 484
Graph minor based technique, 489
Graph subdivision, 485
Gray code, 388–389
Greedy-Then-Oldest (GTO), 544
Ground zero theory, 1327
Grover's algorithm, 239, 734
G-share, 33
Gustafson's Law, 52, 603

H
Halide-HLS, 995, 1010
Hamming distance, 1036
Handling sparsity, in IoT devices, 101
  compressed sparse formats, 103–104
  inner product approach, hardware architecture for, 105–108
  matrix multiplication, approaches in, 102–103
  outer product approach, hardware architecture for, 108–110
  Hard disk drive (HDD), 300
Hardened network transceivers, 423
Hardware, 568, 569
Hardware accelerators, 1037–1038
  DNNs, 1046–1051
  image and video processing applications, 1038–1046
Hardware and software architectures, for video coding, 224–226
  complexity reduction, 227
  DTM for HEVC, 229–232
  low-power memory architectures, 227–229
  workload balancing, for multiple video tiles, 229
Hardware and software (HW/SW), 1380–1385
  partitioning, 958
  performance optimization and validation, 959–960
Hardware aspects, 896, 897
Hardware-based emulation, 953
Hardware complexity, 50
Hardware debugging, 410
Hardware description language (HDL), 843, 850, 949, 953
Hardware emulation, 907, 908
Hardware IFT, 1390
Hardware implementations
  adders, 396–397
  dividers, 398
  multipliers, 397–398
  square root, 398
Hardware in the loop (HiL), 961
Hardware Performance Counters (HPCs), 194
Hardware reprogrammability, 423
Hardware security
  and data protection, 868
  verification, 1390
Hardware/software integration, 867
Hardware vulnerabilities, 1390
Harmonic mean (HM), 634
Harvard architecture, 5
HeteroCL, 1017–1019
Heterogeneous architecture, 842, 864
Heterogeneous CPU cores, 637–638
Heterogeneous image processing acceleration (Hipacc), 995

Heterogeneous integration, 639
Heterogeneous models, 1073
Heterogeneous multiprocessing (HMP), 1071
Heterogeneous pipelining, 1003
Heterogeneous systems, 144–145
Heterogeneous task-level parallelism, 1006
HeteroHalide, 995
Heuristic approaches, 478–483
Hidden Markov models (HMMs), 787
Hidden memory structures, 176
Hierarchical DSE, 932
Hierarchical FPGA, 430, 433
Hierarchical GPU execution model, 537
Hierarchical hardware, 1039
Hierarchical PR, 515
High bandwidth memory (HBM), 456, 459, 613
High Efficiency Video Coding (HEVC), 216–220, 227, 234
  analysis of computational complexity, memory requirements and processor temperature, 220–223
  DTM for, 229
  hardware-software architecture, 225
  memory bandwidth requirement, 223
High level synthesis (HLS), 260, 522, 843, 848, 950, 991–992, 997, 1037
  accelerator design languages, 996
  analysis and optimization of intermediate representation, 853–855
  binding and resource optimization, 857–859
  C-based HLS tools, 993–994
  code generation, evaluation and verification, 863–864
  commercial products and academic projects, history on, 850–851
  dataflow compilers, 994–995
  design flow, 993
  domain-specific architectures, creation of, 865–867
  DSL, 995–996
  FSM controller, 863
  hardware security and data protection, 868
  input specification and intermediate representation, 851–853
  memory architecture, 859–863
  microarchitecture, 855–863
  parallelization techniques, 1007
  programmability and system-level optimization, 867
  scheduling and performance optimization, 855–857
  system-level integration and optimization, 864–865
  traditional high-level synthesis framework, 848–850
High-level partial reconfiguration (HiPR), 522
High performance computing (HPC) data centers, 452, 583
  design-time profiling, heuristics, 584
  fast heuristics, 584
  machine learning, 585
  network technologies, 585
High-resistance state (HRS), 307
High-speed Terasic connector (HSTC), 346
High voltage (HV), 708
Homogeneous data-level parallelism, 1005–1006
Homogeneous pipelining, 1003
Homogeneous synchronous dataflow (HSDF), 1114–1116
Host-based simulation (HbS), 974
Host-compiled simulation, 924
Hot carrier injection (HCI), 284
Hotspots, 611
H-tree, 437–439
Huffman-coded nonzero indication (HNI), 104–106
Hybrid BTB, 34
Hybrid emulation, 964
Hybrid FPGA prototyping, 965
Hybrid mechanistic-empirical modeling, 927
Hybrid models, 1074
Hybrid predictors, 33
Hybrid routing scheme, 333
HyCUBE, 473
Hydra CMP project, 605
HyperOps, 487
Hyperproperties, 1199, 1398–1399
Hypervisor-managed Linear Address Translation, 60

I

IA-32 Basic execution Environment, 58
IA-64 registers, 58, 59
IA32 Architectures, 57
IBMQ London architecture, 1417, 1419
iBTB, 35
IEEE 754, 390–393, 399
IEEE 802.11, 159
IFT verification techniques, 1391
ILP-based approaches, 1001
Image and video processing applications, 1038
  AutoAx methodology, 1038–1042
  Sobel edge detector, 1043–1045

Image signal processor (ISP), 365
Immediate value, 8
Implementation model, 1243
Indirect branches, 8
Indirect unconditional jump, 26
Individual propositional formulas, 1213
Induced subgraph, 485
Induction axiom, 1327
Induction principle, 1212–1213
Induction-variable analysis, 1157
Inductive generalization, 1225
Inductive invariant, 1212
Inductive trace, 1213
Industrial verification, 1314–1315
Infant mortality, 280
Inference rules, 1325
Infinite sequence, 1279
Infinity, 392
Information flow analysis, 1396–1398
Information flow properties, 1395–1396
Information flow tracking (IFT), 1198, 1390, 1391, 1403
Information redundancy, 293, 298
In-order execution, 37
Input feature maps (IFM), 653, 654, 656, 657
Input/Output (I/O) operations, 128
Input reparameterization, 1237
Instantaneous power, 564
Instruction accurate instruction set simulator (IA-ISS), 974
Instruction buffer, 541
Instruction decode, 8
Instruction fetch, 7
Instruction level parallelism (ILP), 5, 36, 43, 469, 536, 596, 597, 601, 605, 606, 608, 640, 903, 1160, 1162
Instruction pipelining, 1160, 1162
Instruction rescheduling, 21
Instruction scheduling, 20, 1160, 1180–1184
Instruction set architecture (ISA), 48, 396, 617, 809, 810, 812, 814, 818–825, 827, 828, 830, 831, 833, 876, 883, 1161, 1200
  binary code analysis, 1335–1336
  formalization, 1333
  formalized ISAs, 1336–1337
  mechanical analysis, 1334–1335
Instruction Set Simulation (ISS), 878, 908, 923
Instruction-set simulator generation, 825–827
Instructions per clock cycle (IPC), 36, 469, 632, 633, 927
Instruction types, 6
Instruction Vulnerability Index (IVI), 289
Instruction-window, 39
Instruction-word parallelism, 1160, 1163, 1180, 1182
Integer arithmetic, 387
  gray code, 388–389
  unary code, 389
Integer linear programming (ILP), 483, 497, 705
Integer multiplication, 1308–1310
Integrated circuit (IC), 278, 562, 566, 596
Integrated GPUs (iGPUs), 139
Intel® arithmetic formal verification, 1272
Intel Itanium™ architecture, 82
Intellectual property (IP), 876, 949
Intel processors, 54, 55
  address modes of ADD instruction, 56
  Math operations in 8086/8088, 56
  register file of 8088 processors, 55
  X86 ISA into 64bits architectures, 57
Intel Quantum SDK, 738
Intel Security Guard Extension (SGX), 173, 184
Intel® TSX Suspend Load Address Tracking, 60
Intel X86-86
  data types, 60
  Intel/Altera, 510, 512, 515
Intended domain, 1330
Inter-application scenario, 933
Interconnect interfaces, 209–210
Interconnection binding, 857
Interconnect topologies, 210
Interference, 610
Interlock, 20
Intermediate representation (IR), 50, 84, 849, 992, 1149, 1153–1154, 1370
Intermittent faults, 281
Internal Configuration Access Port (ICAP), 518, 845
Internet-of-things (IoTs), 90, 92, 469, 808, 1028
  energy harvesting, 92–95
  handling sparsity, 101–110
Interpolation-based model checking, 1223–1225
Interpolation sequence-based model checking, 1222–1223
Interposer, 446, 457–459
Intra-application scenario, 933
Intra-SM resource allocation, 141–143
Intrinsic function, 1165
Inverse-NTT (INTT), 251
IPI Virtualization, 60
ISA agnostic systems binary translation, 85
  virtual machine, 84
  virtual machine and intermediate representation, 85
  ISA-extensible ADL, 827

Island-style FPGA, 431, 433
Isogeny-based schemes, 242
Isomorphic sub-graph (ISG), 744
Iterative floating-point division, 1312
Iterative modulo scheduling (IMS), 480, 1184
Iterative pruning, 1054
Izhikevich model, 327, 336

J
Jitter, 437
Just-in-time compilation (JIT), 1150

K
Kahn process network (KPN), 1003, 1120, 1121
Kaldi toolkit, 787
Kernel density estimation (KDE), 97
Kernel-flatten mapping method, 656
Kernel-splitting mapping method, 656, 657
Key exchange mechanism (KEM), 243
Key performance indicators (KPI), 954, 956, 957, 959
Key performance metric, 1000
k-induction, 1217, 1219–1221
KLEE, 1370, 1374
K-means clustering, 371
Knuth-Yao sampler, 250
K-out-of-N erasure coding, 300

L
Lab-on-a-chip, 688, 689, 701, 707
LaCSNN, 345–346
Language for Instruction Set Architecture (LISA), 818–819
Last-level cache (LLC), 139, 299, 606, 610–613, 616, 621
Last-value predictor, 41
Latching-window masking, 285
Latency hiding, 1009
Latency-insensitive protocols, 862
Latency problem, 1339
Latent section errors (LSEs), 300
Lattice(s), 244, 1392
  computational problems on, 244–245
Lattice-based cryptography (LBC), 243–244
  application specific hardware, 257, 259
  ASIP, 260
  average-case problems on standard lattices, 245–246
  classes of lattices, 246–247
  computationally intensive components of LWE, 248–253
  computational problems on lattices, 244–245
  coprocessors, 254–266
  embedded IoT processors, 256–257
  fault attacks, 268–269
  general optimization strategies, 254
  hardware/software co-design, 260
  high level synthesis, 260
  low resource usage, 255
  optimization strategies for implementation, 261–263
  physical protection, 266–269
  power analysis attacks, 267–268
  R-LWE based PKE scheme, 247–248
  security strength, 255
  side channel attacks resistance, 256
  throughput performance, 256
  timing attacks, 267
Layout versus schematic (LVS), 754
LCC, 1150
Leading Nonzero Detection (LND), 105
Leaky-integrate-and-fire (LIF), 311, 312, 326, 327
Learning methods, 328–329
Learning parity with noise (LPN), 246
Learning with errors (LWE), 243, 245
  computationally intensive components, 248–253
Least laxity first (LLF) scheduler, 133
LeNet5 network, 1062
Lifetime reliability, 280, 287, 289, 578–579
Limited connectivity, 1418
Limited fidelity, 1418
Limited gate-set, 1418
Line-fill buffer (LFB), 184, 185
Linear Address Masking (LAM), 60
Linear time temporal logic, 1207
Liner algebra package (LAPACK), 409
Link, 456, 459
Linker, 1160–1161
Link-time optimization (LTO), 1150, 1161
Linpack, 888
Linux, 630
List scheduling, 481, 1183
Liveness properties, 1207, 1229–1230
  bounded liveness checking, 1232–1234
  FAIR, 1234–1235
  liveness-to-safety conversion, 1231–1232
  SMC with BDDs, 1230
Liveness-to-safety conversion (L2S), 1231–1232
LLVM, 85, 1150, 1152, 1154, 1159–1161, 1170, 1174, 1175
Load, 9

Loadable kernel modules (LKM), 1375–1379
Load/store architecture, 50, 61
Load store queue (LSQ), 572
Load store unit (LSU), 540, 1010
Load value injection (LVI), 187
Local crossbar, 426, 433, 442
Local predictors, 32
Local routing, 428, 433
Local two-level predictors, 32
Logical masking, 285
Logical RAM, 444–446
Logic block (LB), 419, 421–430
Logic element (LE), 421, 424, 429, 430, 445, 458
Logic failures, 306
Loihi, 347
Long-term evolution (LTE), 932
Lookup tables (LUTs), 105, 424, 426–430, 440, 443, 444, 447, 508, 991, 998, 1000
Loop flattening, 495
Loop splitting, 1002
Loop transformation, 1158
Lowering, 1159
Low-pass filter (LPF), 335
Low-power data rate control, 101
Low-power moving object detection, 96
Low-resistance state (LRS), 307, 308

M
Machine epsilon, 387
Machine error codes for processors, 60
Machine learning (ML), 90, 322, 482, 524, 585, 586, 852, 1029, 1044, 1046, 1063
  ANN, architectures for, 349–357
  built-in error tolerance of, 309–310
  classic machine learning, architectures for, 370–371
  neuromorphic computing, architectures for, 324–342
  prominent neuromorphic chips, 343–349
  selective ANN architectures and circuits, 357–370
Machine learning algorithm, 194, 1039–1041
Macro-architecture specification, 956–958
MAGNet, 367
Magnetic tunnel junctions (MTJs), 447
Main memory, 610, 613–614
Main memory policies, 618
  address interleaving, 618
  memory scheduling, 619–621
  row policy, 619
Makimoto's Wave, 53
Manual approximation methods, 1031–1032
Manycore architectures, 405
Manycore CPUs, 615
Mapping approaches, CGRA
  graph theory inspired techniques, 484–491
  heuristic approaches, 478–483
  mathematical optimization techniques, 483
Mapping method for training, 657
  for inference, 654–657
  for training, 657
Mapping symmetries, 931
Market-inspired heuristics, 584
Matrix-matrix multiplication, 291
Matrix multiplication, 102, 109
  inner product-based approach, 102–103
  outer product-based approach, 103
Matrix-vector multiplications, 1431
MaxCompiler, 994
Maximal common subgraph (MCS), 488
Maximum temperature reduction, 574–576
M3D-one designs, 795
M3D stacking technology, 752, 772
Mean absolute error (MAE), 1034
Mean relative error (MRE), 1034
Mean time between failures (MTBF), 286
Mean time to crash (MTTC), 287
Mean time to data loss (MTTDL), 300
Mean time to detection (MTTD), 287
Mean time to failure (MTTF), 287
Measurement error, 728
Mechanical theorem proving, 1197
Mechanistic models, 926, 927
Meltdown, 181
Meltdown attack, 179, 182
Memory, 7, 216, 217, 220, 222, 223, 226–228, 232, 233
  access, 10, 903
  architecture, 846, 859–863
  banking, 1010–1011
  channel, 613, 616, 859
  coalescing unit, 540
  consistency, 614, 626–628
  controller, 436, 455
  failures, 306
  footprint, 50
  hierarchy, 473, 1342–1343
  interfaces, 815
  management, 129
  optimization, 859
  organization in CUDA, 83
  reference disambiguation, 1156
  scheduling, 619–621
  wall, 865

Memory customization, 1007
data vectorization, 1010
decoupled access-execute, 1008
exploiting data reuse, 1007–1008
memory banking, 1010–1011
Memory hierarchy parallelism (MHP), 206, 207
Memory-level parallelism (MLP), 204–208, 210, 211, 618
Memory management, multicore CPU, 615–616
cache coherence, 622–626
main memory policies, 618–621
memory consistency models, 626–628
mitigating interference, 621–622
shared-memory model, 617
Memory scheduling algorithm, 614
Memristive devices, 339, 340
Mescal methodology, 811
Mesh network, 615
MESI cache coherence protocol, 626
Message passing interface (MPI), 1073
Meta-heuristics, 478, 927
Metal oxide semiconductor (MOS) transistors, 335, 336
Micro-architectural attacks, 172
Micro-architectural data sampling attacks, 184–189
Microarchitectural optimization, 614
Micro-architectural poisoning, 187
Microarchitecture, 61, 842
multiple-issue processor, 35–43
pipelining, 12–35
single cycle processor design, 5–12
Microarchitecture properties, 1338
interrupts, out-of-order and speculative execution, self-modifying code, the works, 1340–1342
memory hierarchy, reasoning about, 1342–1343
pipelining, 1338–1340
verification of execution units, 1343–1345
MicroBlaze, 454
Microbumps, 457, 458
Microchannels, 690
Microelectrode cells (MCs), 707, 708
Microelectrode-dot-array (MEDA) biochips, 707, 709, 718
droplet routing and extension, 715–718
hardware implementation, 708–709
MEDA evolution, 711–712
scheduling and placement, 712–714
synthesis methods, 712–718
Microfluidic biochips, 690
DMFBs, 701–707
FBMBs, 690–701
MEDA biochips, 707–718
Micro-operation, 61, 81
Microprocessor/system-on-chip (SoC) architectures, 50–53
Mihalas-Niebur neuron, 337
Millimeter-wave (mmWave), 585
MIMOLA, 816–817
MiniControl, 694
Minimum initiation interval, 1184
MIPS, 5
delayed branch, 80
MIPS-I processor, 62
Mis-predictions per instruction (MPI), 30
Miss status holding registers (MSHRs), 206–207, 613
Miter construction, 1196
Mitigating interference, 621–622
MIV planning stage, 764
Mix/hybrid modes, 49
Mix-precision weight, 665
MMX ISA extension, 75
MobileNet, 310
Model checking, 1195, 1401
Model parameters, 155
Models of computation (MoCs)
advantage, 1112
affine dataflow, 1119
cyclo-static dataflow, 1119
dataflow models, 1109
description, 1110
energy/power consumption, 1128–1129
heterogeneous, 1126–1128
hierarchy, 1119
homogeneous synchronous dataflow, 1114–1116
hybrid mapping, 1132–1133
MAPS project, 1134–1137
multidimension data, 1119–1120
PREESM, 1138–1139
real-time extensions, 1119
semantics, 1110, 1111
SPIDER, 1139
static dataflow models, 1112–1113
static mapping, 1129–1131
synchronous dataflow, 1116–1119
Modern DNNs, 786
Modified list scheduling (MLS), 705
Module lattice-based schemes, 247
Modulo routing resource graph (MRRG), 476–477, 486, 490
Modulo scheduling, 475–476, 1184

Modulo Scheduling with Integrated Register Spilling (MIRS), 487
Monolithic caches, 612
Monolithic three-dimensional integrated circuits, 752
  cascade-2D design, 765, 767
  design-aware partitioning stage, 762
  design libraries, 754
  DNN hardware (*see* Deep neural network (DNN))
  high frequency, M3D power saving at, 760–761
  implementation methodology, 755
  low frequency, M3D power saving at, 758–760
  M3D stacking technology, 772
  power saving trend, 756
  shrunk-2D design flow, 769
  system-level power delivery network analysis (*see* System-level power delivery network analysis)
  technology nodes, 754
Monotonic extension, 1275, 1276
Monotonicity, 1266
Moore, Gordon, 51
Moore's law, 51, 208, 278, 469, 574, 596, 605, 752, 753
Most significant bit (MSB), 658, 659, 673, 675
Motion estimation (ME), 218
MPSoC Application Programming Studio (MAPS), 1134–1137
MSI, 625, 626
MSSQ code, 817
Multi-application workload models, 932–937
Multi-chip modules (MCMs), 639
Multicore architectures, 404, 407, 1164
Multi-core computing
  core-level, 296
  process-level, 296
  redundant multi-threading, 296
  SRT, 296
Multi-core CPUs, 134–137
Multi-level intermediate representation (MLIR), 1150, 1154
Multi-modal Adaptive Collaborative Reconfigurable self-Organized System (MACROS), 297
Multi-objective DSE, 917
Multi-objective optimization, 917
Multi-path delay commutator (MDC), 265
Multiprocessor system-on-chip (MPSoC), 916–918, 921, 925, 926, 928–932, 934, 936, 937
Multi-programmed system, 129
Multi-pumping, 1000
Multi-threading, 1164
Multi-view video coding (MVC), 217
Multicore cache hierarchy, 610
Multicore CPUs, 597, 598, 640
  concurrent processing, 598–606
  controlling CPU core frequency, 630
  coordinating memory requests across cores, 614
  CPU simplification, 607
  DVFS, 608, 609
  evaluations, 631–635
  evolution, 635–639
  four-core, 607
  hardware design, 606–615
  memory management, 615–628
  multiprocessing, 599–600
  multiprogrammed workload performance, 632–634
  multithreaded application performance, 631
  optimizing CPU cores for parallelism, 606–609
  optimizing operating systems, 628–631
  parallel computing hardware, 598–599
  parallelizing programs, 630
  power and energy, 634–635
  scaling to many cores, 614–615
  scheduling threads, 628–630
  sharing caches and main memory, 609–614
  TLP, 601–604
  transistors, 604–606
Multimodal Adaptive Collaborative Reconfigurable self-Organized System (MACROS), 298
Multiple bit-flip fault, 281
Multiple-instruction multiple data (MIMD), 52, 599
Multiple-instruction single data (MISD), 52, 599
Multiple-issue processor, 35–43
Multiple memory channels, 859
Multiple static RTL checks, 1250
Multiplexer, 424, 425, 427, 431, 434, 438, 442, 445
  circuitry, 426
Multiplexors, 7
Multiplier(s), 397, 1032
  array, 430, 447, 449, 450, 453
  circuit, 1291
  routine, 382
Multiply-accumulate (MAC)
  array, 956
  operations, 103, 472

Multiprocessing, 599–600
Multiprogrammed workload performance, 632–634
Multiprogramming, 129
Multithreaded application performance, 631
Mutation, 928
MXNet, 995

N
Naive Bayes classifier (NBC), 370
NanGate FreePDK45 Open Cell Library, 776, 777
Nanos6 data-tracking system, 1094
Nanos6 NUMA-aware scheduling system, 1095–1096
National Institute of Standards and Technology (NIST), 239
Natural language specifications, 810
Negative bias temperature instability (NBTI), 579
Negative channel metal oxide semiconductor (NMOS), 565
Neighbor-to-neighbor (N2N), 472, 473
Nested loop mapping, 494–495
  limited configuration memory, 496
  loop flattening, 495
  polyhedral model, 495
  systolic mapping, 495
Nested PR flow, 516
Network function virtualization (NFV), 523
Network-on-chip (NoC), 210, 300–301, 331–334, 456, 457, 459
Neural architecture search (NAS), 354, 937, 939
Neural network, 312, 885
Neurogrid, 332, 344
Neuromorphic computing, 324, 325
  AER protocol, 334
  biological computing models and learning methods, 324–329
  circuit-level design considerations, 335–342
  framework, 330
  microarchitecture for, 330–334
  neuromorphic core, 330
  NoC, 331–334
Neuromorphic core, 330
  digital asynchronous circuit, 339
  digital synchronous circuit, 339
  memristive devices, 339, 340
Neuromorphic hardware, 305, 306
Neuron(s)
  analog circuits, 335–338
  digital circuits, 338
  models, 326–327
Next state variables, 1208
N-fold data replication, 300
Nios, 453
NIST Post-Quantum Cryptography Standardisation Project, 240–241
nML, 813, 814, 817–818, 1170
N-modular redundancy (NMR), 291
No-flow property, 1407
Noninterference model, 1393
Noisy intermediate-scale quantum (NISQ) computers, 734–735
Noisy systems
  superconducting-specific work, 744–746
  technology agnostic work, 743–744
Non-fracturable LUTs, 428, 430
Non-splitting reshaping-driven detailed routing (NRDR) algorithm, 717
Non-uniform cache access (NUCA), 615
Non-uniform memory access (NUMA), 1071, 1093
  Nanos6 data-tracking system, 1094
  Nanos6 NUMA-aware scheduling system, 1095–1096
  NUMA-aware allocation API, 1093
Non-volatile memory (NVM), 304, 305
  reliability issues in, 306–309
No-operation (nop) insertion, 20–22
Nuclear magnetic resonance (NMR)-based devices, 725
Number theoretic transform (NTT), 251–253
Number-NaN, 392
Numerical accuracy, 387
Numerical precision, 387
Nvidia, 82, 84
NVIDIA Deep Learning Accelerator (NVDLA), 369
NVIDIA GPUs, 147

O
Objective values, 917
Observe-Decide-Act, 523
ODIN, 348
Off-chip accelerators, 209
Off-chip connectivity, 211
Offline replaying phase, 1373
Offline test generation, 1371
OMNX, 85

OmpSs-2 programming model, 1087
advanced dependency system, 1087–1090
commutative dependence type, 1090
concurrent dependence type, 1090
early release of dependencies, 1089
global domain of dependencies, 1088–1090
NUMA support, 1093–1096
optimal task granularity, 1091
reduction type, 1090
semantics of work-sharing tasks, 1092
structured parallelism on many-core processors, 1091–1093
weak dependencies, 1089–1090
work-sharing tasks syntax, 1092
On-chip connectivity, 209
interconnect interfaces, 209–210
interconnect topologies, 210
On-chip memory, 439–447
hierarchy, 474
On-chip SRAM arrays, 790
Online tracing, 1371
Opcode, 7
Open DataFlow (OpenDF), 995
OpenQASM, 1414
Open-source simulators, 904
Open systems interconnection (OSI), 300
OpenCL, 147, 418, 538
OpenCores, 776
OpenMP programming model, 1074–1076
accelerator model, 1085–1086
SIMD support, 1079–1085
tasking model, 1077–1079
worksharing model, 1076–1077
OpenQASM, 737
Operating system (OS), 128–130, 567, 600
Operating system, for multicore CPUs
controlling CPU core frequency, 630
parallelizing programs, 630
scheduling threads, 628–630
Operation bundle, 1160, 1176
Operation per second (OPS), 351
Operator chaining, 998
Optical flow (OF), 97
Optimizations, 1419
OptiML, 995
Ordinary differential equation (ODE), 311
Out-of-order execution, 38, 39
Out-of-order processor, 5, 39, 40, 42
Out-of-order superscalar processor, 175
Output feature map (OFM), 653–657
Over-approximations, 1237–1239
Overconstrained value, 1275, 1276
Overlay, 862, 865
Over-the-top (OTT) media service providers, 232
Oxide-based resistive RAM (OxRRAM), 304, 307
read disturb issue in, 306–308

P

Paging, 129
Parallel and Real-time Embedded Executives Scheduling Method (PREESM), 1138–1139
Parallel computing hardware, 598–599
Parallel efficiency, 632
Parallelization, 992, 993, 998, 1004
heterogeneous task-level parallelism, 1006
homogeneous data-level parallelism, 1005–1006
Parallelizing programs, 630
Parallel linear algebra software for multicore architectures (PLASMAs), 407
Parallel programming models, 1070
constructs in, 1071–1072
hardware models, 1070–1071
OmpSs-2 programming model, 1087–1096
OpenMP programming model, 1074–1086
taxonomy, 1072–1074
XiTAO programming model and runtime, 1096–1103
Parallel random-access machine (PRAM), 52
Parallel speedup, 631
Parameterized and Interfaced SDF (πSDF), 1122–1124
Parameterized set of modes-core functional dataflow (PSM-CFDF), 1125
Pareto construction algorithm, 1045
Pareto dominance, 918, 919
Pareto front/Pareto frontier, 919, 1037, 1040, 1045
Pareto set, 1041, 1042
Parsing, 1155
Partial bitstream, 512, 517
Partial products, 1032
Partial reconfiguration (PR), 459, 509, 511, 512, 514, 517–519, 522, 523, 526
Partially reconfigurable regions (PRRs), 297, 511–513, 515–518
Partial regions, 509
Partitioned scheduling, 135
Partitioned scheduling algorithms, 136
Partition-locked cache (PLCache), 192
Pass-transistor, 424, 426, 433
PathDriver, 695
Path History Table (PHT), 183

PCI express (PCIe), 436, 455, 456
PEAS, 819
Performance, 12
  counters, 636
  estimation, 891
Performance Monitorng Counters (PMC), 567
Performance Per Watt (PPW), 568, 569
Periodic real-time tasks, 131
Peripheral Component Interconnect Express (PCIe interface), 211
Permanent faults, 281
Permutation table, 193
Personal computers (PCs), 599, 604
Phase abstraction, 1237
Phase-change memory (PCM), 308, 309, 340, 580
  cell, 308
  thermal issues, PCM's high voltage operations, 308–309
Phase coupling, 1155
Physical design, 753, 766, 786
Physical registers, 40
Pipeline flush, 29
Pipeline hazards, 18, 1162
  control hazards, 26–35
  data hazards, 19–25
  structural hazards, 35
Pipeline interlock circuit, 22
Pipeline stalls, 20
Pipelining, 13, 998
  dynamically scheduled pipelining, 1002–1004
  operator-level optimizations, 998–1000
  pipelined processors, 16–18
  pipeline hazards, 18–35
  pipeline principle and performance metrics, 12–15
  statically scheduled pipelining, 1000–1002
Placement, 428, 431, 438, 445
PLASMA software stack, 407
Point-of-care diagnostics, 703
Points-to analysis, 1156
Polyhedral model, 495
PolyMage, 995
Polymerase chain reactions (PCR), 691
Polynomial multiplication, 250–253, 263–266
Polynomial number, 246
Population, 928
Population based training process, 938
Posit arithmetic, 394–395
Positional notation, 386
Positive channel metal oxide semiconductor (PMOS), 565
Post-CMOS technologies, 399
Post-CTS optimization, 767
Post-dominator (PDOM), 545
Post-image operator, 1210, 1211
Post-quantum cryptography (PQC), 239
  challenges, 269–270
  code-based, 242
  hash-based, 242
  isogeny-based, 242–243
  LBC, 243–244
  multivariate-based, 242
  NIST PQC, 240–241
Post-route optimization, 767
POWER4, 605
Power analysis attacks, 267–268
Power comparisons, 793
Power consumption, 778, 794
Power converter efficiency, 95
Power delivery networks (PDNs), 753, 776
Power density, 563
Power dissipation, 564, 569
  causes and effects, 564–567
  in multicore systems, 567–568
Power-gating-based active leakage control, 111
  challenges and trade-offs, 112–115
  power gating efficiency learner, 115–116
  self-adaptive power-gating architecture, 116–117
  test-chip and measurement results, 117–119
Power-gating method, 570
Power management, in multicore systems, 563
  AI/ML-based power management, 586
  common power reduction methods, 568–572
  cross-layer approach, 586
  desktop and servers, 580–583
  embedded systems, 572–577
  emerging technologies, 586
  HPC data centers, 583–585
  2.5D/3D systems, 585
Power management unit (PMU), 93, 94
Power-performance-area (PPA), 335
Power reduction methods
  firmware, 569–571
  hardware, 568, 569
  software, 571–572
Power schemes, 582–583
Power wall, 866
Practical circuits, 1401–1402
Pragma, *see* Source-code annotation
Pre-adders, 451
Precision, 387
Predicate abstraction, 1342
Predicated execution, 1158, 1165
Preemptive kernel model (PMK), 142

Presynaptic neuron, 338
Prevention-based countermeasures, 189
Prime+Probe attacks, 176–178
Primitive datatype, 1159, 1164
Primitive function, 1159, 1164
Principal component analysis (PCA), 371
Printed circuit board (PCB) models, 775
Private caches, 610
Probabilistic transfer matrix (PTM), 289
Probability mass function (PMF), 1040
Process design kit (PDK), 754
Process, voltage, and temperature (PVT), 114
Processing, 220, 222, 226, 228, 230, 233
Processing elements (PEs), 351
Processing engines (PEs), 1004
Processing-in-memory (PIM), 148, 352
Processing-near-memory (PNM), 353
Processor architecture, 809, 812, 819, 828, 834
Processor characterization, 883
Processor description languages, 810, 811
Processor IR, 1168, 1170–1171
Processor microarchitecture, 4, 43
Processor selection, 959
Processor subsystems, 453–455
Production rule simulator (PRSIM), 342
Profiling tools, 636
Program control unit (PCU), 1165
Program counter (PC), 6, 49, 71, 544
Programmable accelerators, 809
Programmable array logic (PAL), 423, 424
Programmable delay chains, 436, 439
Programmable IO, 435–436
Programmable logic (PL), 521
Programmable routing, 419, 421, 422, 425, 430–435
Proof-based abstraction (PBA), 1238, 1239
Property directed reachability, 1217, 1225–1228
Property generation, 1410
Proportional-integral-derivative (PID) controller, 101
Pruning, 1054–1055
Pseudorandom sequence (PRS), 119
PTX IR, 85
Public-key encryption (PKE), 243
Pulse latch, 428, 429, 434, 439
PYNQ, 521
Python, 85
PyTorch, 995

Q
Q#, 736, 737
Q-Bit encoding, 481, 482
Quadrant clock, 438
Quadratic LIF (QIF), 326
Quality of service (QoS), 280
Quantization, 1055, 1057–1058
Quantization parameter (QP), 100, 221
Quantum algorithms, 732
  fault tolerant quantum computers, 733–734
  NISQ computers, 734–735
Quantum annealing (QA), 738–739
Quantum approximate optimization algorithm (QAOA), 735
Quantum bits (Qubits), 726, 1414
Quantum charge coupled device (QCCD), 731
Quantum circuit(s)
  alternating approach, 1426
  compilation, 1418–1420
  compilation flow, 1417, 1427–1430
  decision diagrams, 1424–1426
  design flows, 1199
  device under verification, 1421
  diagrams, 1417
  formal verification, 1416, 1423
  golden specification, 1421
  simulative verification, 1416, 1430–1436
Quantum computing, 1414, 1416–1418
Quantum computing architectures
  noisy systems, 743–746
  quantum algorithms, 732–735
  quantum error, 727–729
  quantum gates, 727
  quantum hardware, 729–732
  quantum software, 735–743
  qubits, 726
Quantum error, 727
  crosstalk error, 729
  gate error, 728
  measurement error, 728
  relaxation and dephasing, 728
Quantum gates, 727
Quantum information, 1414
Quantum key distribution (QKD), 270
Quantum programming languages, 1414
Quantum security, 1414
Quantum software, 735
  compilation, mapping and optimization, 739–740
  quantum annealing, 738–739
  quantum instruction sets, 735
  quantum program, 735
  quantum programming languages, 736–738
  quantum software development kits, 736
  superconducting quantum computers, 740–741
  trapped-ion quantum computers, 741–743

Quasi-delay-insensitive (QDI) circuits, 341
Qubit technologies, 730
  spin qubits, 732
  superconducting qubits, 730–731
  trapped-ion qubits, 731–732
QuickRoute, 481
Quiet NaNs (qNaNs), 392
Quipper, 737, 1414

R
Race condition, 129
Radiation, 282, 525
  charge accumulation, 282
Radix, 385
RAM mapping, 444–446
Random matrix, 245
RAP routing protocol, 164
Rate distortion optimization (RDO), 219
Rate monotonic (RM) scheduler, 132
Rathlin image processing language (RIPL), 995
RDSEED instruction, 189
Read-after-write (RAW) dependence, 1001
  dependence, 1001
  hazard, 19–21, 24
Read ports, 7
Real dependencies, 855
Real-time constraint, 1175
Real-time CPU scheduling
  multi-core CPUs, 134–137
  single-core CPUs, 132–134
Real-time flow, 161–163
Real-time operating system (RTOS), 128–132
Real-time scheduling, for CPU-GPU systems
  alternative architectures, 148
  application domains, 146–147
  GPU background, 137–140
  multi-GPU and CPU-GPU scheduling, 143–145
  single GPU, scheduling tasks on, 140–143
  tools and frameworks, 147
Real-time systems, 130, 132
Real-time wired networks
  CAN, 157
  FlexRay, 157–158
  TTEthernet, 158
Real-time wireless networks, 158
  Bluetooth, 158–159
  IEEE 802.11, 159
  sensor, 163–164
  WirelessHART, 160–161
  ZigBee, 160
Receive Control Register (RCTL), 1383
Receiver, 177
Recognition, mining, and synthesis (RMS), 1028
Reconfigurable computing, 418, 423
Reconfigurable modules (RMs), 512, 514, 525
Reconfiguration time, 518
ReConOS, 519
Rectified linear unit (ReLU) function, 788
Recurrent neural network (RNN), 349
Reduced instruction set computers (RISCs), 5, 49, 61–64, 66, 809
Reduced Ordered Binary Decision Diagrams (ROBDD), 1036
Redundancy, 280
  definition, 291
  information redundancy, 293
  spatial/physical, 291
  temporal, 293
Redundant array of inexpensive disks (RAID), 300
Reference model, 1311, 1313
REGIMap, 490
Region of interest (ROI)-aware image processing architecture, 95–96
  conventional target data rate control, 98–99
  data rate control, challenges in, 100
  energy- and content-aware target data rate control, 99
  low-power data rate control, 101
  low-power moving object detection, 96
  noise-robust moving object detection, 97
  spatial ROI-based coding, 98
  temporal ROI-based coding, 97
Register accuracy, 969
Register alias table (RAT), 41
Register allocation, 1160, 1177–1179
  and binding, 857
Register assignment, 1160, 1179–1180
Register binding, 858
Register file (RF), 6, 7, 409, 466, 471
Register-file usage analysis, 904
Register-renaming, 40
Register transfer level (RTL), 567, 843, 850, 949, 950, 953, 955, 959, 964, 966, 968, 971, 973, 1199, 1244–1259, 1261, 1264, 1265, 1268
  co-simulation, 964
  generation and system integration, 863–865
  languages, 990
  simulation, 922
Regression, 1257
Reinforcement learning (RL), 585, 586
Reinforcement learning-based mapping approach, 482

Rejection sampling, 249
Relational STE (rSTE), 1314
Relative error, 386
Relative worst-case error, 1035
Relaxed memory consistency models, 627
Reliability, 286
  estimation, 287–289
  functional, 286
  improvement, 578–580
  lifetime, 287, 289
  metrics, 287
  timing, 286, 289
Reliability-aware HW/HW partitioning, 298
Reliability, GPU
  fault analysis, 557
  run-time error detection and correction, 556–557
Reload phase, 178
Relocation, 1160
Renaming, 40
Reorder buffer (RoB), 43
Representative function, 1339
Reservation station, 40
Residual control, 1165
Resistive-capacitive (RC) parasitics, 752, 766, 771
Resistive random-access memories (ReRAM), 233
ResNet, 310
Resource-aware control encoding
  data rate, 100
  target data rate, 98–100
Resource binding, 858
Resource Director Technology feature, 60
Resource sharing, 857
Response time analysis, 133
Responsive GPGPU execution model (RGEM), 141
Restrict keyword, 1175
Retargetable compiler, 1151–1153, 1167–1184
Retiming, 1236
Return stack buffer (RSB), 35, 183
Reverse in-lining, 1161
Rib clock, 438
Richness and diversity of FPGA IOs, 423
RIFFA, 520
Ring interconnect, 612
Ring lattice-based schemes, 247
Ring-LWE based PKE scheme, 247–248
Ripple-carry adder (RCA), 396–397, 1031
RISC-V, 69, 70
  CHISEL, 822–823
  integer subsystem format, 71
  open specification project, 71
  processors, 71, 909
RISCV, extensions
  atomicity, 72
  atomic operations, 72
  backward compatibility, 73
  floating-point arithmetic, 72
  integer operations with multiplication and division operations, 72
  load-reserved, 72
  relaxed memory model, 72
  single-precision FP, 73
  store-conditional instructions, 72
Robotic vision, 885
ROCm, 147
Rogue in-flight data load (RIDL), 184
Rosetta, 85
Rounding modes, 392–393
Round robin (RR), 129, 141, 536
Router, 433, 456
Routing, 419, 421, 425, 427, 428, 430–435, 767
  port, 427–430, 440–442
  switch, 418, 432–434, 459
  wire, 431–434, 437, 451
Row conflicts, 618
Row decoder, 440, 441
Row interleaving, 618
Row policy, 619
Run-length encoding (RLE), 104

S

Safety properties, 1206
  BMC, 1215, 1216, 1218
  Craig interpolation, 1221
  induction principle, 1212–1213
  interpolation-based model checking, 1223–1225
  interpolation sequence-based model checking, 1222–1223
  k-induction, 1217, 1219, 1221
  property directed reachability, 1217, 1225–1228
  sequence interpolation, 1221–1222
  SMC with BDDs, 1214, 1217
SAGE, 1370
Sampled simulation, 925
Sapphire Rapids microarchitecture, 60
Satisfiability modulo theorem (SMT), 717, 743
Scaffold, 738
Scalable CGRA mapping, 499–500
Scalable timing accuracy, 970
Scenario-based DSE, 933–937

Schedulability
analysis, 133
test, 133, 134
Schedule, place, and route (SPR), 480–481
Scheduling, 855, 857, 858, 863, 1180
policy, 628
quanta, 600
threads, 628–630
Schoolbook multiplication, 264
Scientific computing, 402, 403, 412
CGRAs, 405–406, 410–412
custom architectures, 407
FPGAs, 405, 410
manycore architectures, 405
multi-core architectures, 404, 407
Scoreboard logic, 541
Scratchpad memory (SPM), 471, 473, 574
Security classes, 1392
Security-critical properties, 1409
Security Guarded Extension (SGX), 60
Security lattice, 1392
Security verification, 1390, 1404, 1408
Segmented adders, 1031
Self-adaptive power-gating architecture, 116–117
Self-powered edge-intelligence
energy harvesting, in IoT edges, 92–95
evolution of edge-intelligence and self-powered intelligent computations, 90–92
handling sparsity, in IoT devices, 101–110
power-gating-based active leakage control, 111–119
ROI-aware image processing architecture, 95–101
Self-repair mechanism, 313
Semaphores, 129
Sender, 177
Sense amplifier, 440–442
Sensor nodes (SNs), 304
Sequence interpolation, 1221–1222
Sequential consistency (SC), 627
Sequential equivalence, 1246–1247
Sequential redundancy removal, 1236
Serial transceiver, 436
Serial *vs.* SIMD operation, 74
Service level agreement (SLA), 585
Sesame system-level MPSoC simulation infrastructure, 926
Shared caches, 610
Shared memory, 546–547, 614, 617, 1073
Shell, 456
Shift left, 953, 955, 962, 965
Shor's algorithm, 239, 733
Short coherence times, 1418
Shortest job first (SJF), 129
Shortest vector problem (SVP), 244
Short integer solution (SIS), 245
Shrunk-2D design flow, 768, 769
Signaling NaNs (sNaNs), 392
Signal processing, 303
Signed-magnitude representation, 388
Silicon, 421
compilers, 850
Simple AND gate, 1402–1403
Simple dual port RAM, 444
Simplified AER-based communication system, 334
Simulated annealing, 478
Simulation, 878, 902, 906
Booleans and undefined values, 1275–1276
circuit properties, 1281
circuit simulation and undefined values, 1276–1278
mathematical model of circuit properties, 1282–1283
mathematical model of circuit simulation, 1279–1281
relation, 1339
speed, 969
testing, 1194
Simulation-based verification, 1403–1404
Simulation scope control
property triggers, 1289–1291
reachable state invariants, 1295–1297
scope reduction by triggers, 1293–1295
Simulative verification
classical circuits, 1431–1432
equivalence checking flow, 1436–1437
matrix-vector multiplications, 1431
stimuli generation schemes, 1432–1436
two quantum circuits, 1430
Simultaneous and redundant threading (SRT), 296
Simultaneous multi-threading (SMT), 296
Single-bit-error-correcting (SEC), 298
Single bit-flip fault, 281
Single-cell analysis, 703
Single-core computing, 294–296
Single-core CPU, 132–134, 597, 604
Single-core processor, 48, 652
Single-cycle core, 16
Single cycle processor design, 5
processor control unit, 11–12
processor data path, 6–9
Single event effects (SEEs), 524
Single event latchup (SEL), 282
Single event upset (SEU), 285, 525

Single GPU, scheduling tasks
inter-SM resource allocation, 142–143
intra-SM resource allocation, 141–142
memory transfer between device and host, 143
Single-instruction multiple data (SIMD), 52, 73, 74, 78, 408, 599, 1005, 1006, 1079, 1158, 1163
function vectorization, 1084–1085
loops, 1082–1084
vectorization, intrinsics and semi-automatic vectorization, 1080–1081
Single-instruction multiple thread (SIMT), 137, 139, 532, 533, 536, 540, 542, 544–545, 554
Single-instruction single data (SISD), 52, 137, 599
Single-nibble-error-correcting (SNC), 298
Single-port RAM, 444
Single scheduling algorithm, 136
Single-threaded objects, 1332
Single-thread sequential programs, 596
SLLI (logical left shift), 71
Small outline DIMM (SODIMM), 299
Snoopy cache coherence, 624
Sobel edge detector, 1043–1045
Soft bus, 436, 455
Soft error, 459
Soft error rate (SER), 285, 579, 580
Soft processor, 452–455
Software
aspects, 898–900
bring-up, 953, 960
data forwarding, 571
pipelining, 1182–1183
regression testing, 962–963
task migration, 571
task scheduling, 571
testing, 954, 963, 979
verification, 1358
Software-based redundant multi-threading (SRMT), 296
Software development kits (SDKs), 953
Software-driven functional verification, 963
hybrid emulation, 964
hybrid FPGA prototyping, 965
RTL co-simulation, 964
system-level power analysis, 965–966
Software Topology Address (STA), 1098
Software use-cases, virtual prototyping, 960–961
early software development, 962
non-intrusive and platform-level debug, 961
scripting, 961
software regression testing, 962–963
tracing and analysis, 961
SOLAR, 698
Solid-state drive (SSD), 300
Sortex, 699
Source-code annotation, 1175
Source operands, 7, 9
Source register 1 (sreg1), 7
Source register 2 (sreg2), 7
SPAC Coprocessor, 64
SPADES, 521
SPARC, 63, 64
Spatial CGRA, 472
Spatial redundancy, 291, 292
Spatial ROI-based coding, 98
Specialization, 1164
Specification language, 809
Specification model, 1243
Spectre attack, 179
Speculative adders, 1031
Speculative execution, 1158
Speech recognition, 786
Speed, 952
SPEED routing protocol, 164
SpGEMM operations, 108
CSC format, 106
HNI format, 106
Spike-based backpropagation supervised learning method, 328
Spike-driven synaptic plasticity (SDSP), 328, 329, 331, 348
Spike-timing-dependent plasticity (STDP), 327–329, 331, 337, 338, 649
Spiking neural networks (SNN), 323, 324, 328, 330, 334–336, 340, 343, 345, 348, 371
Spine clock, 438, 439
SpiNNaker, 343
Spin qubits, 732
Spin-torque transfer magnetic random access memory (STT-MRAM), 667, 669, 671, 681
Spin-transfer-torque magnetic RAM (STTMRAM), 340
Sporadic real-time tasks, 131
Square root, 398
SRAM-based configuration memory, 508
SRLL (logical right shift), 71
SSD controller SoC, 978–979, 982
accurate virtual prototype, 980–982
loosely timed virtual prototype, 979–980
Stall generation, 22
Standard benchmarks, 884, 885
Berkeley Design Technology, 892

CoreMark, 889, 890
Dhrystone, 888–889
EEMBC, 892
Embench, 890, 891
estimating processor performance, 884, 886, 887
Linpack, 888
SPEC CPU, 891
Whetstone, 888
Standard/random lattice-based schemes, 246
State explosion problem, 1204, 1207
State machine, 1205
Statically scheduled pipelining, 1000–1002
Static branch prediction, 30
Static dataflow models, 1112–1113
Static dissipation, 566
Static rail analysis, 778, 781
Static random access memory (SRAM), 298, 299, 574, 580, 610, 649, 662, 667–669, 671, 672, 681–683
cells, 418, 424, 426, 433, 440–442, 452, 459
leakage energy, 95
Static region (SR), 509, 511–513, 515, 516
Static scheduling, 1002
Static single-assignment (SSA), 851, 852, 1154, 1156
Static techniques, 1193, 1194
Statistical simulation, 925
Stochastic quantization, 665, 666
Stop-go task scheduling algorithms, 576
Store, 9
buffers, 627
Store-to-load forwarding, 187
Straight-line code, 1080
Stratix II, 426, 427
Stratix III, 445, 446
Stratix IV, 446, 451
Stratix V, 428, 438, 445
Streaming multiprocessors (SMs), 139, 540
Streaming SIMD Extensions (SSE), 75
Stream processing, 409
Strength reduction, 1159
Stride value predictor, 42
Structural hazards, 18, 35
Structured pruning, 1055–1057
Stuck-at fault, 281
Subgraph-homeomorphism-based techniques, 486–487
Subnormal numbers, 391
Substitute-And-SIMpllfy (SASIMI), 1033
Substrate, 459
Subthreshold circuits, 335
Subthreshold leakage, 566
Sum-of-absolute-difference (SAD), 218
Superconducting quantum computers
compilation and optimization, 741
coupling constraints and need for SWAP operation, 740
Superconducting qubits, 730–731
Superconducting-specific work
application specific compilation, 745–746
crosstalk mitigation, 744–745
leveraging extended native gates, 745
Super-linear speedup, 603
Super-pipeline, 25, 35
Superscalar, 36, 1163
CPU, 174
Supervised learning, 328
Superword level parallelism, 1080
Support vector machines (SVM), 370
SuSy, 996
Sweeney-Robertson-Tocher (SRT) type iterative division algorithm, 1311
SW/HW codesign, 84
Swing modulo scheduling, 1184
Switch block, 421, 431, 432, 442
Switched-capacitor circuits, 337
Switch statement, 1158
Symbolic Boolean, 1283, 1284
Symbolic computation, 1283–1284
Symbolic execution, 1367–1369
Symbolic Execution Engines (SEE), 1367
Symbolic instantiation, 1283
Symbolic lift, 1283
Symbolic model checking (SMC), 1195
with BDDs, 1214, 1217, 1230
Symbolic simulation, 1197, 1270–1271
and FEV, 1273
and FPV, 1273
as formal verification, 1271–1272
mathematical model of, 1287–1288
practical considerations, 1288–1289
symbolic computation, 1283–1284
symbolic values, 1284–1287
and theorem proving, 1274
Symbolic state, 1287, 1288
Symbolic stimulus, 1288
Symbolic trace, 1287
Symbolic trajectory evaluation (STE), 1272, 1315
Symbolic values, 1284–1287
Symbolic variables, 1283
Symmetrical Multi-threading (SMT), 174
Symmetric multiprocessing (SMP), 404
Synapse models, 327
Synaptic efficacy, 340

Synchronization, 129
Synchronous dataflow (SDF), 497–498, 1116, 1119
Synchronous Parameterized and Interfaced Dataflow Embedded Runtime (SPIDER), 1139
Synopsys ARC cores, 909
Synthesis, 418
System as a service (SAS), 86
Systematic methodology for Automatic Logic Synthesis of Approximate circuits (SALSA), 1032–1033
SystemC, 950–952, 967
SystemC 1.0, 951–952
SystemC 2.0, 952
SystemC Modeling Library (SCML), 971, 976, 977
SystemC Transaction Level Modeling Standard, 967–969
  approximately timed modeling style, 970–971
  extended AT, 971
  extended loosely timed modeling style, 970
  loosely timed modeling style, 969
System-level design, 916, 1247
System-level power analysis, 958, 965–966
System-level power delivery network analysis
  analysis methods, 777
  dynamic rail analysis, 781
  frequency- and time-domain analysis, 782–785
  static rail analysis, 778
  system-level power delivery network modeling, 775
System matrix, 1422, 1423
System-memory interfaces, 815
System of difference constraints (SDC), 856, 1001
System-on-chip (SoC), 139, 202–203, 211, 469, 635–636, 1193, 1201, 1239, 1380–1385
  accelerators, 208–209
  architecture optimization, 960
  balanced processor architectures, 205
  CPU memory parallelism, 206–208
  CPU types, 204–205
  design and verification, 949–950
  processor, 203–208
System under test (SUT), 1367
Systolic array(s), 1004
  architecture, 862, 866
Systolic mapping, 495

T
Tag store, 611
Target-address, 7
Target binary program (TBP), 1372
Target technology, 844–846
Task granularity, 1091
Tasking model, 1077–1079
Task instance, 131
Task-level parallelism, 1164
Task mapping strategies, 573
Task migration, 571
Task parallelism, 1071
Task scheduling, 571
Technology agnostic work
  measurement error mitigation, 744
  noise-aware qubit mapping, 743–744
Technology-level laws, 51
Technology scaling, 278, 842
Temperature, 223, 224, 231, 234
Temporal decoupling, 969
Temporal redundancy, 293
Temporal ROI-based coding, 97
TEMU, 1370
Tensilica TIE, 819–820, 831
Tensor processing unit (TPU), 471, 809
TensorFlow, 995
TensorFlow-32 (TF32), 396
Ternary Content-Addressable Memory (TCAM), 523
Test selection phase, 1373
Texture memory, 546
Theorem prover, 1197
Theorem proving
  ACL2 preliminaries, 1324–1332
  analog systems, 1358
  analysis of microarchitecture properties, 1338–1345
  concurrent protocols, 1358
  formalization and analysis of (simplified) x86, 1345–1358
  ISA, 1332–1337
  and microprocessor assurance, 1322–1324
  software verification, 1358
Thermal design point/thermal design power (TDP), 570, 576, 577, 605, 609
Thermal management, 574–577
Thinker series, 360–361
Thread-level parallelism (TLP), 469, 562, 601–604, 606, 628
Thread of control, 1164
3D microarchitecture, 568

3D NoC routing scheme, 333
Throughput, 13
Through-silicon vias (TSVs), 752
Thumb, 64, 66
  and ARM register maps, 68
  instruction formats, 65
Thumb-I, 66
Thumb-II, 66
Tianjic, 348
TianoCore utility programs, 1374
Time dependent dielectric breakdown (TDDB), 284
Time-division multiple access (TDMA), 157
Time-sharing, 600
Time-Triggered Ethernet (TTEthernet), 158
Timing
  accuracy, 952
  analysis, 768
  flows, 1395, 1406
  reliability, 280, 286, 289, 294
TLM-2.0, 967
Top of stack (TOS), 545
Total ionizing dose (TID), 282
Total store ordering (TSO), 627
Tournament predictor, 33
Tow's complement arithmetic, 388
Trace-based analysis methods, 1402
Trace capture phase, 1373
Trace-driven simulation, 924
Trace properties, 1199, 1398–1399
Trace scheduling, 1184
Trace selection phase, 1373
Traditional circuit simulator, 1271
Traditional computer-assisted theorem proving, 1274
Transaction-based equivalence, 1247–1248
Transaction-level modeling (TLM), 924, 967, 973
  approximately timed modeling style, 970–971
  base protocol, 971
  extended AT, 971
  extended loosely timed modeling style, 970
  levels of abstraction, 973
  loosely timed modeling style, 969
  peripheral components, 976–977
  processor models, 974–976
  transport interface, blocking, 969
  transport interface, non-blocking, 971
Transient faults, 281, 282
Transient instructions, 174
Transient memory, 128
Transient micro-architectural attacks, 173
  countermeasures, 189
  detection-based solutions, 194
  prevention-based countermeasures, 189–193
Transition relation, 1208, 1209
Transition system, 1208–1209, 1212, 1213, 1217
Transitive micro-architectural attacks, 181
Transmeta, 86
Trapped-ion quantum computers
  compilation and optimization, 742–743
  shuttle operation, 741–742
Trapped-ion qubits, 731–732
Triple-bit-error-detecting (TED), 298
Triple-DES (3DES), 238
Triple modular redundancy (TMR), 291, 294, 525, 557
Tri-state buffer switch, 434
Trivial arithmetic property, 1266
True-data dependency, 19
True dual-port memory, 442
TrueNorth, 346
Trusted execution environments (TEE), 173
Truth tables, 11
Tseitin transformation, 1209
T2S-Tensor, 996, 1010
Turing, Alan, 51
Turing Machine, 51
Turn-around time, 952
TVM, 996
Two-bit saturated counter, 31
2D IC design flow, 767
2D-mesh routing scheme, 332
2-dimensional transform, 881
2D-tree routing scheme, 332
Two-level branch predictors, 32
Two-level global BTB, 34

U

Ucode model, 1352–1353
Ultrascale+ architecture, 439
Unary code, 389
Unconditional direct jumps, 6
Unconditional indirect jumps, 6
Undefined reg initial values, 1251
Undetected disk errors (UDEs), 300
Unified shader core, 535
Uniform cache access, 612
Uniform memory access (UMA), 1071

Uniform recurrence equations (UREs), 996
Unit is the last place (ULP), 387
Universal Boolean functional vectors, 1293
Universal Power Format, 965
Universal Verification Methodology (UVM), 964
Universal weakening, 1300
Uop semantic functions, 1352
User-defined microcode programming, 81
Utilization, 997

V
Vector addition kernel functions, 538
Validation, 1192
Value prediction, 41
Variable-length instruction, 1163
Variational quantum eigensolver (VQE), 734
Variation-aware qubit allocation (VQA), 743
Variation-aware qubit mobility (VQM), 743
VAX architecture, 54
Vax machines, 54
Vector architectures, 78
Vectorization, 408
Vector machines 1972–1996, 79
Vector operations, 49
Vector predication, 1165
Vector processor, 454, 455
Verifiable interfaces, 1193
Verification, 1192, 1194, 1271, 1272, 1322, 1324, 1329, 1331
  automated, 1192, 1193
  bit level model checking algorithms, 1195
  C-to-RTL equivalence checking, 1195–1197
  decode block, 1355–1356
  exec block, 1353
  of execution units, 1343–1345
  exhaustive, 1192
  flow, 1302–1304
  formal, 1194
  information flow analysis, 1198
  mechanical theorem proving, 1197
  of quantum circuit design flows, 1199
  symbolic simulation, 1197
  versatile binary-level concolic testing, 1198
  xlate/ucode blocks, 1356–1357
Verification methodology
  assertions for proofs, 1253
  assume guarantee, 1256
  blackboxing, 1253–1254
  bug hunting, 1256
  case splits, 1254–1255
  constraints on inputs, 1251–1252
  convergence, 1254–1256
  coverage, 1256
  C++–RTL mapping, 1250–1251
  cutpoints, 1255, 1256
  debug, 1254
  dynamic quick checks, 1253
  dynamic weakening, 1256
  full proofs, 1253
  linting checks in RTL, 1250
  linting checks on C++/SystemC, 1250
  modularity, 1254
  regression, 1257
  RTL front end compilation, 1250
  SystemC/C++ front end compilation, 1248–1249
  wrapper around C++/SystemC model, 1249–1250
Verilog code, 1205, 1206
Verilog-to-Routing (VTR), 420, 421
Versal, 457
Versatile binary-level concolic testing, 1198, 1367–1368
  infrastructure of versatile binary-level concolic testing, 1371–1375
Versatile Tensor Accelerator (VTA), 354, 370
Versatile Video Coding (VVC), 216
Very long instruction word (VLIW), 5, 37, 601, 1163
  architectures, 81
VEX, 76
VGG11 network, 1062
Victim tag array (VTA), 550
Video codecs, 216–217
Video coding, 224–226
  complexity reduction, 227
  DTM for HEVC, 229–232
  low-power memory architectures, 227–229
  workload balancing, for multiple video tiles, 229
Video compression, 885
VINE, 1370
Virtex-II, 447
Virtex-4, 449, 450
Virtex-5, 427, 452
Virtex-7, 458
Virtualization, 571
Virtual machines (VMs), 571, 583, 584
Virtual memory, 130
Virtual processing units (VPU), 957
Virtual prototyping, of processor based platforms
  architecture analysis, 956–960
  historic background, 951–952

hybrid use-cases, for software-driven functional verification, 963–967
SoC design and verification, 949–950
software use-cases, 960–963
SSD controller SoC, 977–982
transaction level virtual prototypes, 967–977
use-cases for early architecture analysis, 955
verification continuum, 952–954
Virtual-VDD, 112–114
Vivado HLS, 1010
VLSI, 354, 369, 371
Voltage regulator (VR), 583
Voltage regulator module (VRM), 775
Voltage scheduling, 582
Von Neumann model, 5, 48
decode stage, 49
execution stage, 49
interpretation of the instruction, 49
memory stage, 49
write back (commit) stage, 49

W
WAGE, 666
Warp scheduling algorithm, 551
Weak ordering, 628
Wearout-based faults, 280
Weighted speedups, 633
Whetstone, 888
Whitebox models, 926
Whole-program optimization, 1160
WirelessHART, 160–161
Wireless image sensor nodes, 92
Wireless sensor networks (WSNs), 304
Wordline, 440
Worksharing model, 1076–1077
Work stealing queues (WSQ), 1102
Worst-case execution time (WCET), 286
Write-after-read (WAR) hazards, 40, 41
Write-after-write (WAW) hazards, 40, 41
Write-back, 7, 10
Write port, 7
Write transient forwarding (WTF), 187

X
x86, 1345
candidate instruction, 1353–1354
decode block, verification of, 1355–1356
design considerations, 1347–1349
exec block, verification of, 1353
instruction specification function, 1346
machine state, 1345
run function, 1346
scope, 1349
step function, 1346
ucode model, 1352–1353
verifying x86 instruction implementations, 1350–1358
xlate/ucode blocks, verification of, 1356–1357
X86-64 architectures, 60
Xilinx Runtime Library (XRT), 520
XiTAO data-parallel interface, 1100
XiTAO internals, 1102–1103
asynchronous data parallel mode, 1101
configuring the runtime, 1103
explicit DAG programming, 1096–1098
locality-aware moldable mapping, 1099
software topology mapping, 1098–1099
synchronous data parallel mode, 1101
XiTAO programming model and runtime, 1096
X-pessimism, 1275
XSAVE instruction, 78

Y
Y-chart
approach, 957
methodology, 920
Y-chart-based DSE, 920–922

Z
Zero-overhead loop, 1158
ZigBee, 160
Zombieload, 173, 181, 186
ZyPR, 519