Design and Modeling of Specialized Architectures

A DISSERTATION PRESENTED

BY

YAKUN SOPHIA SHAO

TO

THE SCHOOL OF ENGINEERING AND APPLIED SCIENCES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN THE SUBJECT OF

COMPUTER SCIENCE

HARVARD UNIVERSITY
CAMBRIDGE, MASSACHUSETTS
MAY 2016

Thesis advisor: Professor David Brooks and Gu-Yeon Wei Yakun Sophia Shao

Design and Modeling of Specialized Architectures

Abstract

Hardware acceleration in the form of customized datapath and control circuitry tuned to specific applications has gained popularity for its promise to utilize transistors more efficiently. However, architectural research in the area of specialization architectures is still in its preliminary stages. A major obstacle for such research has been the lack of an architecture-level infrastructure that analyzes and quantifies the benefits and trade-offs across different designs options. Existing accelerator design primarily relies on creating Register-Transfer Level (RTL) implementations, a tedious and time-consuming process, making early-stage, design space exploration for specialized architecture designs infeasible.

This dissertation presents the case for early-stage, architecture-level design methodologies in specialized architecture design and modeling. Starting with workload characterization, the proposed ISA-independent workload characterization approach demonstrates its capability to capture application's intrinsic characteristics without being biased due to micro-architecture and ISA artifacts. Moreover, to speed up the accelerator design process, this dissertation presents a new modeling methodology for quickly and accurately estimating accelerator power, performance, and area without RTL generation. Aladdin, as a working example of this methodology, is 100× faster than the existing RTL-based simulation, and yet maintains accuracy within 7% of RTL implementations. Finally, accelerators are only part of the entire System on a Chip (SoC). To accurately capture the interactions across CPUs, accelerators, and shared resources, we developed an integrated SoC simulator based on Aladdin to enable system architects to study system-level ramifications of accelerator integration.

The techniques presented in this thesis demonstrate some initial steps towards early-stage, architecture-level infrastructures for specialized architectures. We hope that this work, and the other research in the area of accelerator modeling and design, will open up the field of specialized architectures to a wider range of researchers, unlocking new opportunities for efficient accelerator design.

Contents

Title Page **1**

Copyright **2**

Abstract **iii**

Table of Contents **ix**

List of Figures **xiv**

Previous Work **xv**

Acknowledgements **xvi**

Dedication **xx**

1 Introduction **1**

1.1 What is an accelerator? 1

1.2 A tale of two scalings 2

1.2.1 Moore scaling 2

1.2.2 Dennard scaling 4

1.3 The combination of Moore and Dennard scaling 5

1.3.1 Moore + Dennard—where we were 6

1.3.2 Moore Scaling only—where we are 6

1.3.3 Dennard only—where we are unlikely to be 8

1.3.4 A future without scaling: “The winter of despair” 8

1.4 To live without scaling: “A spring of hope” 10

1.4.1 Why not architectural scaling? 11

1.4.2 Specialization makes a difference 12

1.5 Challenges in Specialized Architecture Design 15

1.6 Thesis Contributions 15

1.6.1 Accelerator Workload Characterization (Chapter 3). 16

1.6.2 Accelerator Pre-RTL Modeling (Chapter 4). 16

1.6.3 Accelerator-System Co-Design (Chapter 5). 17

1.7 Thesis Organization 18

2 Background and Related Work 19

2.1 Accelerator taxonomy 19

2.1.1 Accelerators that are part of the pipeline. 22

2.1.2 Accelerators that are attached to cache 26

2.1.3 Accelerators that are attached to the memory bus 30

2.1.4 Accelerators that are attached to the I/O bus 34

2.2 Standard RTL Design Flow 34

2.3 High-Level Synthesis 35

2.4 Putting it Together 39

3 WIICA: ISA-Independent Workload Characterization for Accelerators 40

3.1 Introduction 40

3.2 Motivation 42

3.2.1 Stack Overhead. 42

3.2.2 Complex Operations. 44

3.2.3 Calling Convention. 44

3.3 Methodology and background 45

3.3.1 Compiler's IR. 46

3.3.2 ISA-Dependent. 47

3.3.3 Sampling. 47

3.3.4 Benchmark Suite. 48

3.4 Wordload Characteristics Analysis 48

3.4.1 Compute 48

3.4.2 Memory 53

3.4.3 Control 57

3.5 Putting it all together 59

4 Aladdin: Pre-RTL, Power-Performance-Area Accelerator Modeling 61

4.1 Introduction 62

4.2 Background and Motivation 63

4.2.1 Accelerator Design Flow 63

4.2.2 Accelerator Design Space 64

4.2.3 State-of-the-art Accelerator Research Infrastructure 65

4.2.4 Contributions 66

4.3 The Aladdin Framework 66

4.3.1 Modeling Methodology 66

4.3.2 Optimization Phase 68
4.3.3 Realization Phase 70
4.3.4 Integration with Memory System 75
4.3.5 Limitations 76
4.4 Aladdin Validation 77
4.4.1 Validation Flow 77
4.4.2 Applications 78
4.4.3 Validation 81
4.4.4 Algorithm-to-Solution Time 82
4.5 Case Study: GEMM Design Space 82
4.5.1 Execution Time Decomposition 83
4.5.2 Accelerator Design Space 85
4.5.3 Resource-Sharing Effects in Heterogeneous SoC 86

5 gem5-Aladdin: Accelerator-System Co-Design 88

5.1 Introduction 88
5.2 Motivation and Background 90
5.2.1 Co-design: A Motivating Example 90
5.2.2 Typical CPU-Accelerator Communication 91
5.3 Modeling infrastructure 92
5.3.1 Overview 93
5.3.2 Accelerator Modeling 94
5.3.3 DMA Engine 94
5.3.4 Caches and Virtual Memory 95
5.3.5 CPU-Accelerator Interface 95

5.3.6 Performance Validation . . . 95
5.4 Memory System Opportunities . . . 97
5.4.1 Primary design considerations . . . 97
5.4.2 DMA Optimizations . . . 98
5.4.3 DMA Evaluation . . . 100
5.4.4 Cache-Based Accelerators . . . 103
5.4.5 Cache Evaluation . . . 104
5.5 Accelerator Design Choices . . . 105
5.5.1 DMA vs. Caches . . . 106
5.5.2 Design Decision Comparison . . . 108
5.6 Related Work . . . 111

6 Conclusions and Future Directions **113**

References **132**

Listing of figures

1.1 In 1965, Gordon Moore predicted that the number of transistors per integrated circuit would double every year (later revised to every two years)[115]. . . . 3

1.2 The minimum cost per transistor can be reached by balancing the density of transistors and the yield rate of fabrication[115]. 3

1.3 Four relations of Moore and Dennard scaling. 5

1.4 Transistor size[34]. 6

1.5 Transistor count[34]. 6

1.6 Cost per transistor[116]. 6

1.7 Supply voltage stops scaling[59]. 7

1.8 Clock frequency stops scaling[59]. 7

1.9 Achievable speedup with respect to the parallel portion of the program according to Amdahl's Law[22]. 8

1.10 Achievable multi-core speedup across technology generations compared to Moore's law scaling[64]. 8

1.11 Intel historical technology scaling trend and projections[9,10]. 9

1.12 Cost per transistor stops scaling[2]. 10

1.13 Performance increase is more than technology scaling[59]. 11

1.14 Specialization makes a difference. 12

1.15 An annotated die photo of an Apple A8 (iPhone 6) SoC. The yellow box on the lower right is a dual-core CPU, and the red box on the lower left is a quad-core GPU. More than half of the die area is dedicated to non-CPU, non-GPU blocks (white boxes), most of which are application-specific accelerators. The original die photo is from Chipworks[4]. 13

1.16 Die photo analysis across generations of Apple's SoCs. Left: Die area breakdown. Right: Number of specialized IP blocks. 14

1.17 Future Heterogeneous Architecture. 14

2.1 DySER pipeline[74]. 23

2.2 Neural processing unit[65]. 24

2.3 An example of 10x10 architecture[47]. 24

2.4 Convolution engine system overview[129]. 25

2.5 An example of the Tensilica processor pipeline[149]. 26

2.6 An example of Hwacha vector accelerators[122]. 27

2.7 CHARM architecture[54]. 27

2.8 A C-core-enabled system[159]. 28

2.9 SNNAP[117]. 29

2.10 The overall Memcached architecture[103]. 29

2.11 Hardware acceleration of database operations system overview[40]. 30

2.12 Area/power/delay characteristics of Q100 tiles compared to a Xeon core[165]. 31

2.13 LINQits hardware overview[48]. 31

2.14 Accelerator store design[110]. 32

2.15 The Sonic Millip3De Hardware Overview[136]. 32

2.16 Diannao Accelerator[43]. 33

2.17 A 2-core system w/ HARP integration[164]. 33
2.18 (a) Catapult FPGA block diagram. (b) Manufactured board. (c) The server that hosts the FPGA board.[128]. 34
2.19 Synthesis flow. 35
2.20 High-level synthesis landscape. 36

3.1 WIICA Overview. 42
3.2 The percentage of stack instructions of total dynamic instructions for 32-bit and 64-bit x86 binaries. 43
3.3 Instruction breakdown of complex (top three bars) and single (bottom bar) operation instructions. 45
3.4 The instruction breakdown for x86, IR and Simplified-IR (S-IR). 49
3.5 The instruction breakdown for x86, IR and Simplified-IR (S-IR). 50
3.6 x86 51
3.7 IR 51
3.8 Cumulative distribution of the number of unique opcodes of 179.art. The intersecting lines show the number of unique opcodes that cover 90% of dynamic instructions. 51
3.9 Number of unique opcodes to cover 90% of dynamic instructions. “All” represents the global superset. 52
3.10 Number of unique static instructions to cover 90% of dynamic instructions. 52
3.11 Number of unique memory addresses to cover 100% of dynamic memory accesses. 54
3.12 Number of unique memory addresses to cover 90% of dynamic memory accesses. 54

3.13 Memory address entropy of x86, x86 without stack, and IR traces. Lower values indicate more regularity in the access stream. 56
3.14 Example of Local Entropy 57
3.15 179.art 57
3.16 255.vortex 57
3.17 Local memory entropy as a function of low-order bits omitted in calculation. A faster dropping curve indicates more spatial locality in the address stream. 57
3.18 Number of unique branch instructions to cover 90% of dynamic branches. . 58
3.19 Branch entropy per workload. Lower values imply better branch predictability. 58
3.20 Comparison of five ISA-independent metrics across SPEC benchmarks, ordered by the area of the polygon. The lower right kiviat plot provides the legend, and smaller values indicate more regularity in the metric. 59

4.1 GEMM design space w/ and w/o memory hierarchy. 64
4.2 The Aladdin Framework Overview. 66
4.3 C, IR, Resource Constrained DDDG, and Activity. 73
4.4 Cycle-by-Cycle FU and Memory Activity of FFT. 75
4.5 Validation Flow. 77
4.6 Unoptimized vs. Tuned Scan. 78
4.7 Performance (top), Power (middle), and Area (bottom) Validation. 80
4.8 Energy Characterization of SHOC. 81
4.9 GEMM Time and Power Decomposition 83
4.10 GEMM design space. 85
4.11 Design Space of GEMM without and with contention in L2 cache. 86

5.1 Isolated and co-designed EDP optimal design points for stencil3d. 90

5.2 Data movement overheads on MachSuite. 92

5.3 An example SoC that can be modeled using gem5-Aladdin. The table shows the set of design parameters that we swept and their values; this is just a small subset of what can be changed. 93

5.4 Error between Zedboard and gem5-Aladdin cycles. 96

5.5 A demonstration of the DMA latency reduction techniques, using `reduction` as an example workload. 99

5.6 Cumulatively applying each technique reduces the additional cycles spent on DMA, with some benchmarks seeing more benefit than others. After applying all techniques, increasing parallelism through loop unrolling reduces compute cycles until near-complete overlap is achieved, causing performance to saturate. 101

5.7 Effect of datapath parallelism on cache-based accelerator performance. . . 104

5.8 Power-performance Pareto curves for DMA- and cache-based accelerators. EDP optimal design points are shown as stars. Benchmarks are ordered left-to-right, top-down by preference for a DMA-based vs. a cache-based memory system. 106

5.9 Comparison of accelerator microarchitectural parameters across four design scenarios. The vertices of the kiviat plots represent the number of datapath lanes, SRAM sizes, and local memory bandwidth, normalized to the maximum across the four scenarios for each benchmark. 109

5.10 Energy delay product improvement of co-designed accelerators in different scenarios compared to the isolated designs. The design parameters of each optimal design points are illustrated in Figure 5.9. 110

Previous Work

Portions of this dissertation appeared in the following:

Y. Shao and D. Brooks, "ISA-Independent Workload Characterization and Its Implications for Specialized Architectures", Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS), April 2013.

Y. Shao and D. Brooks, "Energy Characterization and Instruction-Level Energy Model of Intel's Xeon Phi Processor", Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED), September 2013.

B. Reagen, Y. Shao, G. Wei and D. Brooks, "Quantifying Acceleration: Power Performance Trade-Offs of Application Kernels in Hardware", Proceedings of the International Symposium on Lower Power Electronics and Design (ISLPED), September 2013.

Y. Shao, B. Reagen, G. Wei and D. Brooks, "Aladdin: A Pre-RTL, Power-Performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures", Proceedings of the International Symposium on Computer Architecture (ISCA), June 2014.

B. Reagen, R. Adolf, Y. Shao, G. Wei and D. Brooks, "MachSuite: Benchmarks for Accelerator Design and Customized Architectures", Proceedings of the International Symposium on Workload Characterization (IISWC), October 2014.

Y. Shao, B. Reagen, G. Wei and D. Brooks, "The Aladdin Approach to Accelerator Design and Modeling," Proceedings of IEEE Micro's Top Picks in Computer Architecture Conferences (TopPicks), May 2015.

Y. Shao and D. Brooks, "Research Infrastructures for Hardware Accelerators", Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers, November 2015.

Y. Shao, S. Xi, V. Srinivasan, G. Wei and D. Brooks, "Co-Designing Accelerators and SoC Interfaces using gem5-Aladdin," Proceedings of the International Symposium on Microarchitecture (MICRO), October 2016.

Acknowledgments

Finally, this is the end. I still remember when I first started my PhD, David sent me to a summer school on computer architecture and compiler design in Barcelona. Enjoying the amazing architecture and wonderful food there, I was joking with David that if grad school is like this, I would love to be in grad school forever. After spending seven years in grad school, I am relieved that it does not come true:). Looking back, it has been an amazing journey since then, and I am grateful for all the help and support I have received over the years. I would not be who I am today without so many special people in my life.

First and foremost, I am extremely fortunate to have two amazing advisors to interact with on a daily basis: David Brooks and Gu-Yeon Wei. They have a tremendous impact on who I am today. As a fresh-out-of-undergrad student with close to zero knowledge about how to do good research, David and Gu courageously took me in their group and taught me everything from thinking critically in research to aligning objects in PowerPoint. I thank them for seeing potential in me that I did not know I had, for teaching me how to do good research and effectively communicate my ideas to others, and for always being available to chat whenever I need advice. They heard me out when I was confused, and have guided me through all the ups and downs of my PhD. I could not have asked for better advisors other than David and Gu.

I would like to thank Margo Seltzer for serving on my dissertation committee. I first met Margo when I took her operating system class during my first semester at Harvard. It was an interesting but also challenging course for me back then. I am grateful for her support and confidence in me not only during the class but also throughout my PhD. I also thank her for the detailed feedback she gives me on my work, which has enriched my dissertation as well as my vision for future directions. I also want to express my gratitude to Eddie Kohler and Stephen Chong for serving on my qualification committee. I appreciate all the early feedback that they have given me to improve my dissertation research.

I am also thankful for many of the mentors I have outside Harvard over the years. I thank Victor Lee and Pradeep Dubey for hosting my internship at Intel Labs. I also thank Viji Srinivasan, Hillery Hunter, Pradip Bose, and Jeff Burns for hosting my two internships at IBM T.J. Waston Research Center. I am also grateful to IBM for awarding me the prestigious IBM PhD Fellowship. Moreover, I would like to thank Margaret Martonosi for encouraging me to write the Synthesis Lecture book. I also want to thank Joel Emer for his support and feedback on my work. I am looking forward to working with him at NVIDIA Research afterwards.

My PhD journey would not be complete without the past and present amazing members of the VLSI-Arch group. I especially thank Glenn Holloway for his invaluable help whenever I run into any machine problems. I am also grateful for the editing he has done for almost all my papers and writings. I will miss the mid-night phone calls and emails from Glenn about me using up vlsiarch space. I would like to thank Mike Lyons for helping me settling down when I first joined the group. I also thank him for the great coffee he made, for the trips we took together for good Asian food, for Om Nom, and for the free Dropbox space he has given me. I also want to thank Vijay Janapa Reddi for his confidence in me. I never really worked with VJ so far, but he always tells me that he sees potential in me. Maybe he just say that to everybody, but I took it seriously. I thank him for being a cheerleader for all my accomplishments. I also want to thank Simone Campanoni for introducing me to the compiler world with his awesome ILDJIT tool. His passion and persistence towards research have been a source of inspiration that keeps me motivated. Moreover, I would like to thank my two major collaborators, Brandon Reagen and Sam Xi, for the wonderful collaborations we have during the past years. They have brought so much energy to the projects and powered me through all the tough times during the process. I also thank many other members of the group: Meeta, Krishna, Jud, Kevin, Emma, Saketh, Bob, Paul, Svilen, Rafael, Ankur, Wonyoung, Mario, Simon, Saekyu, Hyunkwang, Siming, Tao, and Silvia. I have shared so many great memories with them, and each and every one of them has made it fun to come to the lab every day.

I would also like to thank my friends and mentors outside the lab. Iris, thank you for always finding the most amazing restaurants no matter whether we are in Boston, Beijing or Taipei. I will definitely miss all the fun lunch conversations we had together. Jun, thank you for being the sweetest roommate in our lovely apartment during the past years. Dongxing, thank you for being a great friend and mentor with whom I can share my thoughts. Pamala, thank you for working with me on my English when I first came to the US. I also want to thank my undergraduate advisors, Wen Xu, Bingtao Ruan, and Mansun Chan for keeping in touch with me throughout my PhD and always having faith in me since we first met.

A special thank you goes to my wonderful boyfriend, Zhongnan Fang. It is hard to believe that it has been almost nine years since we first met in Hangzhou. Since then, we have been to so many beautiful places and shared so many wonderful memories together. I thank him for always being there for me, for believing in me no matter what happens, and for all the adventures and laughs he has brought into my life. It has been a fun journey, and I am looking forward to sharing more wonderful experiences with you!

Last but not the least, I thank my parents for their unconditional love and support throughout my life, even when we are continents away. I owe all my accomplishments to them.

Dedicated to my family, friends, and mentors.

"It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it was the epoch of belief, it was the epoch of incredulity, it was the season of Light, it was the season of Darkness, it was the spring of hope, it was the winter of despair."

Dickens, 1859

1 Introduction

The era of heterogeneous computing is upon us. Heterogeneity comes in many forms, including domain-specific processors and application-specific accelerators. Almost all major semiconductor vendors have chips that include accelerators, big or small, for a variety of applications. Research in this space has grown in popularity, and there is a vibrant community of researchers in the fields of computer architecture and VLSI-CAD that seek to embrace this new era. But what exactly is an accelerator and why do we need accelerators now? In the following sections, we define accelerators and discuss how fundamental changes in the semiconductor industry have brought about the emergence of this new era of hardware accelerators.

1.1 What is an accelerator?

An accelerator is a specialized hardware unit that performs a set of tasks with higher performance or better energy efficiency than a general-purpose CPU. Examples of accelerators include digital signal processors (DSPs), graphics processing units (GPUs) and fixed-function application-specific integrated circuits (ASICs), such as video decoders. The use of accelerators is not a new idea; the deployment of floating point co-processors in the 1980s marked one of the early adoptions of accelerators, and accelerators have been a common feature of system-on-chip (SoC) architectures for embedded systems for decades. However, accelerator research and deployment in mainstream computer architectures has not been embraced until very recently. This raises the question: why accelerators, now?

To answer this question, let us first look at the interaction between computer architecture and semiconductor technology. Computer architecture sits as a layer between the semiconductor fabrication and circuit technology used to produce chips and the high-level system software and applications that run on them. Hence, every major architectural breakthrough is deeply rooted in the advance of underlying technology and the quest for better performance or energy efficiency for applications. In the rest of this chapter, we discuss the historical scaling trends that have driven the growth of the semiconductor industry and how disruption in these trends has led to the growing popularity of accelerators.

1.2 A tale of two scalings

The semiconductor industry has been driven by two scaling laws: Moore's Law and Dennard scaling. It is these two scaling trends that have resulted in the popularity of CMOS technology and subsequent advances in computing technology over the past several decades.

1.2.1 Moore scaling

The semiconductor industry has recorded impressive achievements since 1965 when Gordon Moore published the observation that would become the industry's guiding standard for the subsequent five decades[115]. Moore's law states that the number of transistors that can economically be fit onto an integrated circuit doubles every two years, demonstrated in the well-known plot shown in Figure 1.1. This simple plot set the pace for progress in the semiconductor industry, although Moore's scaling prediction is purely an empirical observation of technological progress.

Moore's law is about more than just shrinking transistor sizes and better integration capability. It is fundamentally a cost-scaling law. Moore derived the law as he was ultimately interested in shrinking transistor costs. Figure 1.2 is also from Moore's original paper, and is an often overlooked observation that is every bit as important as transistor density trends:

"For simple circuits, the cost per component is nearly inversely proportional

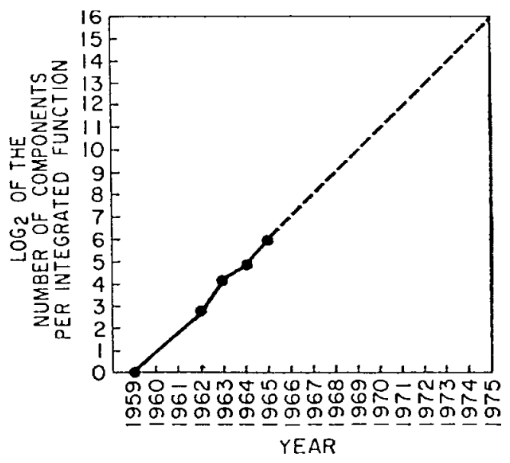


Figure 1.1: In 1965, Gordon Moore predicted that the number of transistors per integrated circuit would double every year (later revised to every two years)[115].

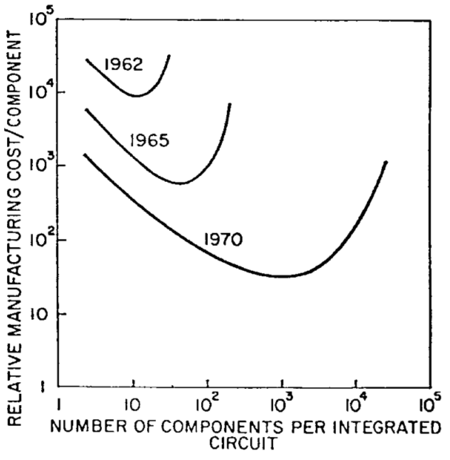


Figure 1.2: The minimum cost per transistor can be reached by balancing the density of transistors and the yield rate of fabrication[115].

> to the number of components, the result of the equivalent piece of semiconductor in the equivalent package containing more components. But as components are added, decreased yields more than compensate for the increased complexity, tending to raise the cost per component. Thus there is a *minimum* cost at any given time in the evolution of the technology."

What Moore observed is that the cost per transistor depends on two factors. One is the density of transistors that we can cram onto a single chip, and the other one is the yield rate of the fabrication, i.e., the fraction of working chips on a wafer. On the one hand, increasing the number of transistors per silicon area makes each individual transistor cheaper on average. On the other hand, as chip design becomes more complex with more transistors, the chances of a chip being rendered inoperable by defects found on the wafers also increase, driving up the cost per transistor.

Over the years, Moore's law has driven the semiconductor industry to lower the cost per transistor by scaling the sizes of transistors and wafers together with a combination of other innovative "circuit and device cleverness" to increase the yield[137]. Specifically, scaling the sizes of transistor and wafer affects the transistor cost in the following ways:

- Transistor size—the smaller the better since we can cram more transistors on the same die area;

Device or Circuit Parameter	Scaling Factor
Device dimension t_{ox}, L, W	$1/k$
Doping concentration N_a	k
Voltage V	$1/k$
Current I	$1/k$
Capacitance eA/t	$1/k$
Delay time per circuit VC/I	$1/k$
Power dissipation per circuit VI	$1/k^2$
Power density VI/A	1

Table 1.1: Dennard scaling detailed how device and circuit parameters follow the scaling of transistor dimensions[60].

- Wafer size—the larger the better since we can produce more chips from a fixed number of processing steps. Also, empirically defects are more likely to occur at the edge of a wafer, so a larger wafer also means smaller defective densities.

1.2.2 Dennard scaling

Moore's analysis only stated that we need to make transistors smaller but it did not touch on how to make this happen. It is Dennard scaling that outlined how we make smaller transistors at each technology generation. Robert Dennard addressed this in his seminal paper on metal-oxide semiconductor (MOS) device scaling in 1974[60]. In that paper, Dennard showed that when voltages are scaled along with transistor dimensions, a device's electric fields remain constant, and most device characteristics are preserved. Table 1.1 summarizes how transistor and circuit parameter changes under ideal scaling conditions, where k is a unit-less scaling constant. In a new technology generation, the transistor dimensions, e.g., gate oxidide thickness (t_{ox}), length (L), and width (W), become smaller compared to the previous generation by a factor of $1/k$. As transistors get smaller, they switch faster, use less power but achieve the same power density. Dennard's scaling has set the roadmap for the semiconductor industry for each generation of process technology, with a concrete transistor scaling formula to move each generation forward.

1.3 The combination of Moore and Dennard scaling

As we discussed earlier, Moore's scaling fundamentally is about cost scaling: the more transistors that can be packed into a given silicon area, the *cheaper* it is to fabricate a transistor. On the other hand, Dennard's scaling is about performance scaling. It demonstrated how transistors have *better* delay and power characteristics as they get smaller. Eventually, every scaling theory faces two possibilities: to continue or to end. By combining the two possibilities with Moore's and Dennard's scaling theories, we end up with a combination of four scaling trends, shown in Figure 1.3.

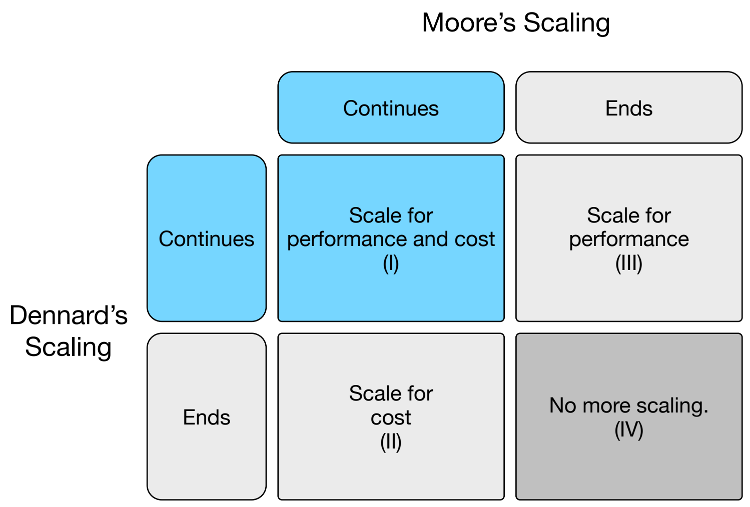


Figure 1.3: Four relations of Moore and Dennard scaling.

1. Both Moore and Dennard scalings continue—Scale for both performance and cost (Region I). This is the ideal scaling region, which the semiconductor industry enjoyed up until the 2000s, resulting in both faster and cheaper transistors.

2. Moore scaling only—Scale for cost (Region II). The state of the semiconductor industry since the mid 2000s: Dennard scaling stops but we still have cheaper transistors per generation.

3. Dennard scaling only—Scale for performance (Region III). This scaling region has not been realized in an economically practical way.

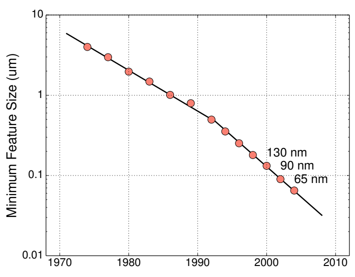


Figure 1.4: Transistor size[34].

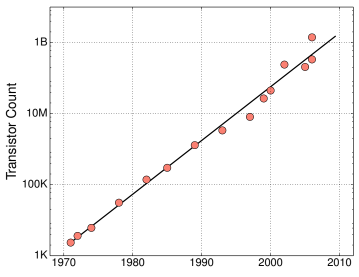


Figure 1.5: Transistor count[34].

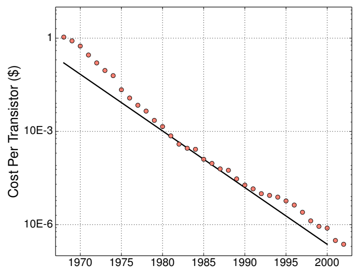


Figure 1.6: Cost per transistor[116].

4. No more scaling—Scale for nothing (Region IV). CMOS technology will become a commodity, likely resulting in lower profit margins for fabrication companies. In this case, there will be little motivation to devote any efforts to continue scaling until a new transistor technology emerges to displace end-of-CMOS devices.

We will discuss each of the four trends and how they have shaped the evolution of the computing industry.

1.3.1 Moore + Dennard—where we were

The semiconductor industry reaped the benefits from scaling in Region I for a long while, where both Moore and Dennard scaling provided nearly ideal benefits to chip designers. Figure 1.4 shows that the minimum feature size scaled consistently over a 30 year period from the mid-1970s to the mid-2000s, closely tracking with Dennard scaling projections and providing designers with smaller, faster transistors. At the same time, Moore scaling, represented by a doubling of the number of transistors every two years (Figure 1.5), led to an exponential decrease in cost per transistor (Figure 1.6).

1.3.2 Moore Scaling only—where we are

By the middle part of the 2000s, the semiconductor industry realized that scaling trends had created a major problem with power consumption that made it difficult to economically cool microprocessors. The fundamental reason is that while the industry was successful in matching Dennard scaling projections in producing smaller transistors, the chip supply voltage had not kept pace with the theoretical projections.

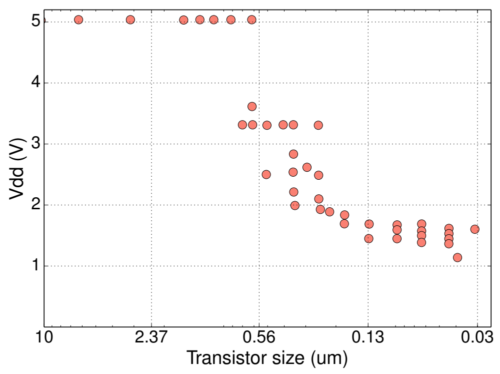


Figure 1.7: Supply voltage stops scaling[59].

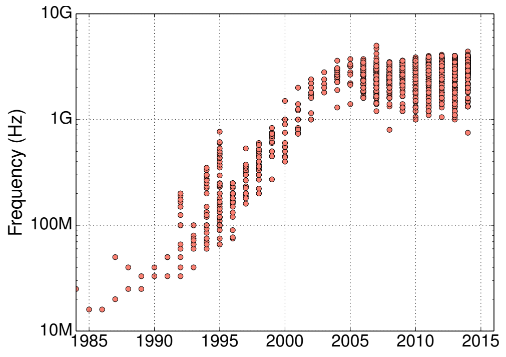


Figure 1.8: Clock frequency stops scaling[59].

Figure 1.7 shows historically how supply voltage scales with transistor feature size. The data here is collected from CPU DB[59], an open database that aggregates detailed processor specifications of more than 790 processors from 17 manufacturers over the past 40 years. We notice that from the 0.13 *um* generation on, supply voltage scaling has slowed down, which is tightly coupled to the lack of threshold voltage scaling in devices to counter sub-threshold leakage current. At the same time, we also observed an abrupt end to the clock frequency scaling era in the early 2000s, illustrated in Figure 1.8. The reason behind this is power. As voltage scaling stops, in order to keep power density constant, the industry has moved away from deep-pipelined, high-frequency machines and is now designing machines with relatively fixed frequencies[80,83,150].

After loosing the benefits of clock frequency scaling, with the increasing number of transistors provided by Moore's scaling, the microprocessor industry adopted multicore architectures that use many simpler processors to keep power density low while increasing the aggregated performance of the entire chip through thread-level parallelism. However, the multicore approach does not fundamentally extend Dennard's performance scaling.

First, as illustrated by Amdahl's Law, the overall speedup is still heavily limited by the sequential portion of the application[22]. Figure 1.9 shows the ideal speedup with respect to the number of cores for applications with different parallelism. For extreme parallel workloads, e.g., more than 95% portion of the program is parallelizable, the achievable speedup for 256 cores is less than 20 $\times$. Second, the worsening energy and speed scaling of transistors further limits the number of transistors that can be powered on at

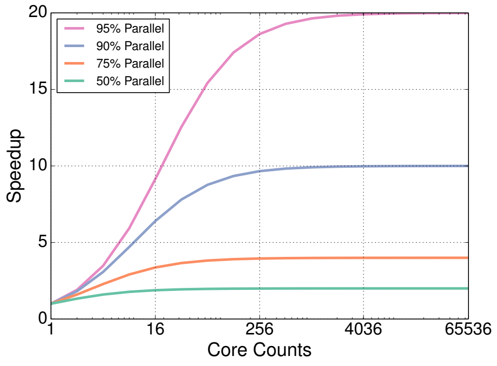


Figure 1.9: Achievable speedup with respect to the parallel portion of the program according to Amdahl's Law[22].

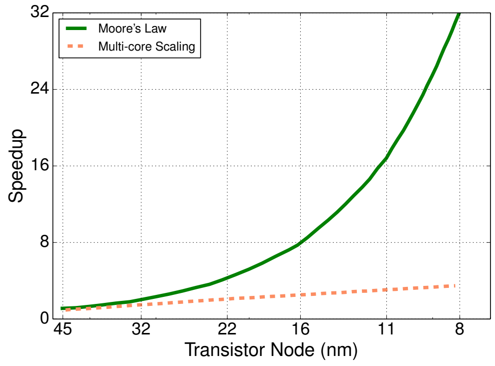


Figure 1.10: Achievable multi-core speedup across technology generations compared to Moore's law scaling[64].

the same time, leading to transistor under-utilization (i.e., dark silicon[64]). Figure 1.10 illustrates the overall achievable speedup over technology generations. Due to the parallelism and power limitation, a significant gap exists between what is achievable with multi-core scaling and what is expected from Moore's Law (i.e., doubling performance with every technology generation). The question for architects now is: what's next?

1.3.3 Dennard only—where we are unlikely to be

Although it is unlikely that we will enter a regime of Dennard-only scaling, it could exist if there were a new device technology that scaled in performance and energy, but not in the economic dimension of Moore scaling. For the sake of discussion, such a technology would likely be confined to niche applications that need ultra high performance or low energy consumption and are not cost sensitive. Thus, it is likely that only a small subset of semiconductor players would be interested in exploring such technologies, as the capital and R&D expenses of maintaining scaling would be hard to overcome.

1.3.4 A future without scaling: "The winter of despair"

A return of Region I scaling is unlikely in the medium term given device technology trends and projections over the next decade. At the same time, traditional Moore scaling is already measurably slowing down. Figure 1.11 shows Intel's historical technology scaling trend based on the release date of the first microprocessor in each technology

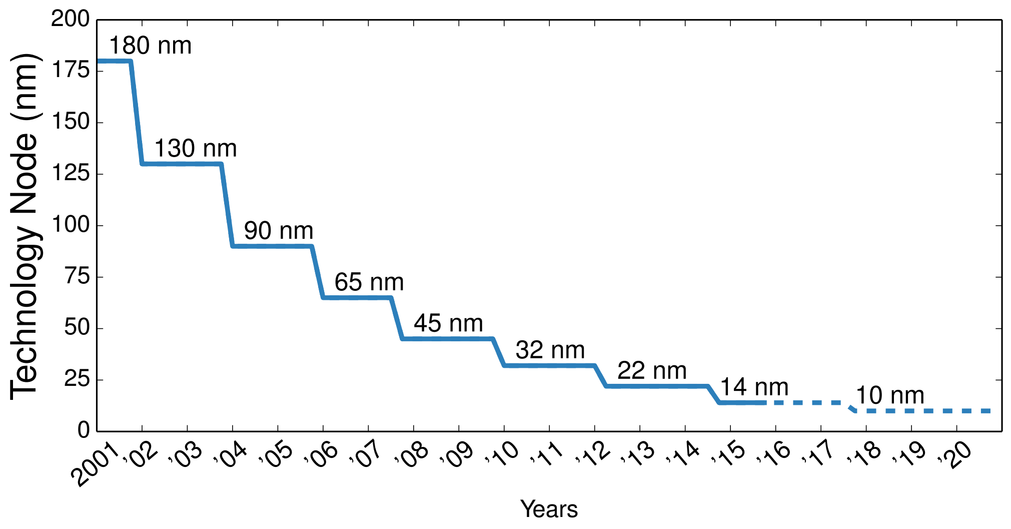


Figure 1.11: Intel historical technology scaling trend and projections[9,10].

node, along with projections for the arrival of the 10 *nm* node. Over the past decade, Intel has been following its famous tick-tock development cycle: first releasing processors with an existing architecture but a new technology node (tick), and then the following year releasing processors with a new architecture on the then-mature technology node (tock). In this way, Intel can release new products every year, alternating architecture and technology improvements. However, the introduction of Intel's 14 *nm* process in the fourth quarter of 2014 was a disruption of this rhythm as it was delayed for half of a year beyond the original projection of the second quarter. Moreover, in July 2015, Intel announced that the 10 *nm* node will not be ready until the second half of 2017, which is another significant delay[10]. In this case, there will be three generations of products using the same 14 *nm* technology.

Intel's latest delay is part of a larger trend in semiconductor manufacturing. Switching to a new process node is getting more complex and more expensive than ever. There has been a drastic increase in the following two categories of fabrication costs.

First, the R&D cost for developing next generation CMOS transistors is increasing rapidly. Innovations like FinFETs have allowed the semiconductor industry to continue to scale transistor feature sizes. However, the fabrication process is getting significantly more complex due to techniques such as multi-patterning[157,171]. Such costs overcome lithography bottlenecks at the expense of more fabrication steps, expensive tools, and

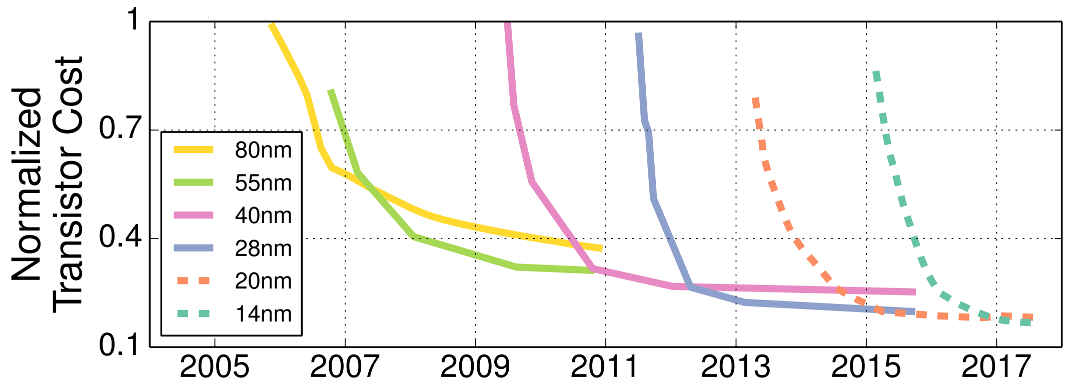


Figure 1.12: Cost per transistor stops scaling[2].

higher mask design costs. Several semiconductor companies have become fabless in the past several years due to the increasing cost of maintaining state-of-the-art fabrication facilities[7], and it is likely that we will see even more consolidation in the industry.

Second, it is also getting harder to increase the wafer size. Larger wafer size can lower fabrication costs by increasing the number of dies per wafer and providing better yields. However, the cost of equipment grows significantly with larger wafer sizes. Migrating from today's 300 *mm* wafers to full-scale 450 *mm* fabrication lines will cost $10 billion to $15 billion, and thus this transition is is not expected to occur until 2020[14].

Figure 1.12 shows the cost of transistors over different technology generations in recent years. The data, provided by Nvidia[2], shows that there is a minimal projected cost-benefit after migrating past 28 *nm* technology, as the projected curves have normalized cost to 28 *nm*. Thus, scaling to smaller feature sizes will no longer provide an economic benefit for fabless IC companies. If this comes to pass, it will effectively mean the end of the economic scaling projection of Moore's Law.

1.4 To live without scaling: "A spring of hope"

The computing industry has begun to adjust to the loss of Dennard scaling, and the loss of Moore scaling is on the near-term horizon. Without either kind of scaling, there is the risk of stagnation in the overall computing industry. One possible outcome of this is that transistors will become a commodity with potentially lower profits for fabrication companies. This will likely lead to additional consolidation in the semiconductor industry, e.g., the recent Intel-Altera and Avago-Broadcom acquisitions[3,8], and fabrication companies

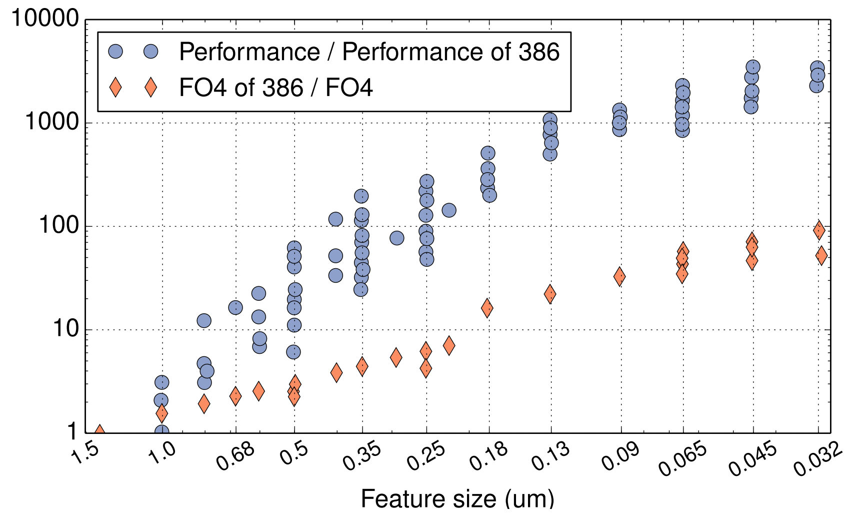


Figure 1.13: Performance increase is more than technology scaling[59].

will rely on "More-than-Moore" technologies to provide differentiation[23].

At the same time, technology disruptions often mean new opportunities, and in this case, there is a significant opportunity for innovation at the design and architecture level. Companies will increasingly differentiate their products based on vertically integrated solutions that leverage new applications mapped to innovative hardware architectures. In this context, application- and domain-specific hardware accelerators are one of the most promising solutions for improving computing performance and energy efficiency in a future with little benefit from device technology innovation.

1.4.1 Why not architectural scaling?

Even with both Moore and Dennard scaling, architecture-level innovations have already contributed significantly to the performance increase of computing systems. Figure 1.13 illustrates the normalized performance of multiple generations of processors since Intel's 80386 microprocessor, also from CPU DB[59]. To estimate the performance of a processor if it were manufactured using a newer technology without micro-architecture changes, the authors use the delay of an inverter driving four equivalent inverters (a fanout of four, or FO4) to quantify the speedup achieved through frequency scaling. We see that roughly half of the overall performance improvement is due to faster transistors; the other half is mostly from architectural innovations. With the increasing number of tran-

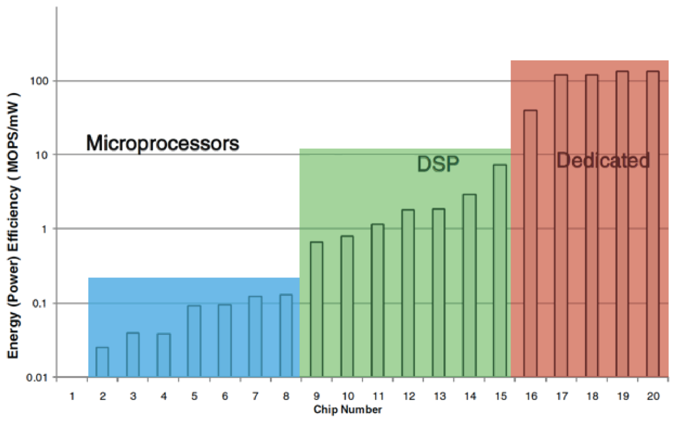


Figure 1.14: Specialization makes a difference.

sistors on a chip, computer architects have been working hard to make use of these transistors for better performance. Examples include superscalar and out-of-order scheduling to increase instruction-level parallelism, better cache hierarchies to overcome memory bottlenecks, multi-core architecture to harness thread-level parallelism, and specialized units, like SIMD, to increase the performance of applications with data-level parallelism.

Figure 1.13 suggests that to overcome the loss of FO4 scaling over the next decade, computer architects will need to deliver an additional order of magnitude performance improvement for the computing industry to enjoy the same amount of performance differentiation that we achieved over the past decade. However, it is unlikely that we will see these same increases in performance and efficiency for traditional general-purpose cores going forward, as many of the above techniques have reached the plateau of performance improvement and already incurred unwanted power overheads.

1.4.2 Specialization makes a difference

Hardware acceleration in the form of datapath and control circuitry customized to particular algorithms or applications has surfaced as a promising approach, as it delivers orders of magnitude performance and energy benefits compared to general-purpose solutions. Figure 1.14 shows the energy efficiency comparison between general-purpose processors, DSPs, and dedicated application-specific accelerators[172]. The data was collected from 20 different chips across different architectures. These chips were originally published at the International Solid State Circuits Conference (ISSCC) between 1998 and

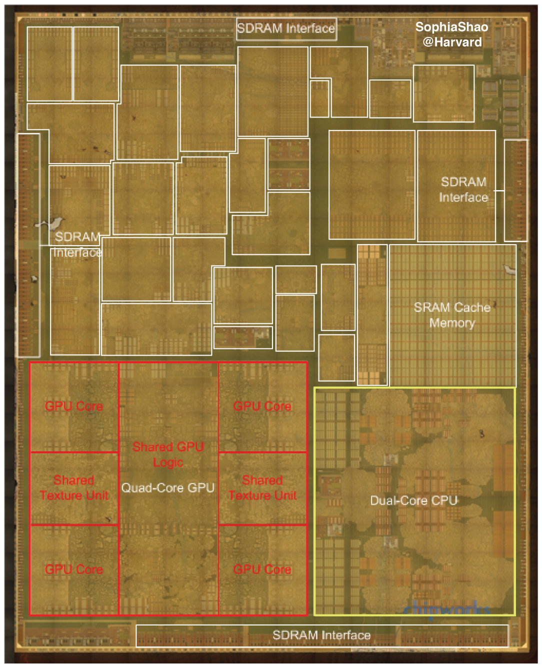


Figure 1.15: An annotated die photo of an Apple A8 (iPhone 6) SoC. The yellow box on the lower right is a dual-core CPU, and the red box on the lower left is a quad-core GPU. More than half of the die area is dedicated to non-CPU, non-GPU blocks (white boxes), most of which are application-specific accelerators. The original die photo is from Chipworks[4].

2002. Compared to general-purpose processors, customized processors like DSPs deliver from 10× to 100× more energy efficiency, while dedicated application-specific accelerators are 1000× more energy efficient.

Specialized architectures were first adopted in the power-constrained mobile market. Figure 1.15 shows a die photo of an Apple's A8 SoC. We notice that the CPU (the yellow box on the lower right) takes only 15% of the die area, while the GPU (the red box on the lower left) takes 25% of the die area. More than half of the SoC area is dedicated to non-CPU, non-GPU blocks, most of which are application-specific hardware accelerators. This area allocation is interestingly common. Analysis of die photos[146] from three generations of Apple's SoCs: A6 (iPhone 5), A7 (iPhone 5S), and A8 (iPhone 6), identifies the same trend, as illustrated in Figure 1.16 (left). In addition, we also observe an increasing diversity in these specialized blocks. Figure 1.16 (right) shows that the number of specialized IP blocks across five generations of Apple SoCs has been growing consistently over the past decade.

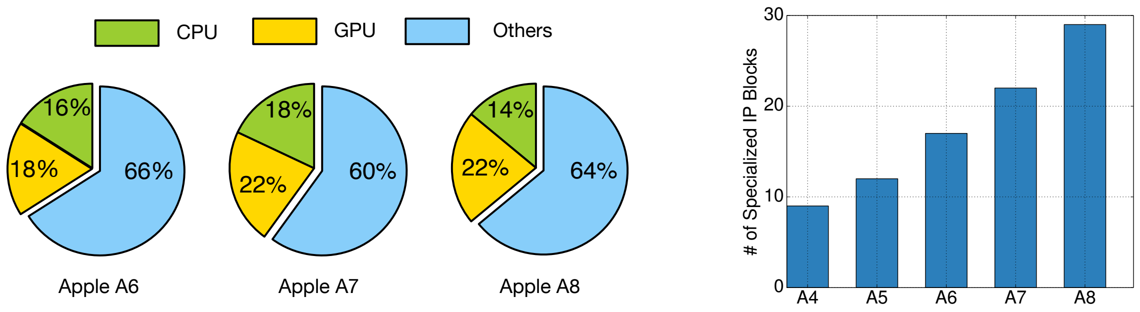


Figure 1.16: Die photo analysis across generations of Apple's SoCs. Left: Die area breakdown. Right: Number of specialized IP blocks.

Moreover, the need for specialized architectures is no longer solely associated with the mobile market. The lack of device scaling has motivated the need for specialization in virtually every type of computing system, ranging from mobile SoCs, desktops, and even data centers[88,111,128,143]. The natural evolution of the increasing number of accelerators will lead to a growing volume and diversity of customized accelerators in future systems. We envision future heterogeneous architectures would include CPUs, GPUs, and a diverse set of specialized accelerators, all of which are connected through shared memory and network on chip (NoC), as illustrated in Figure 1.17.

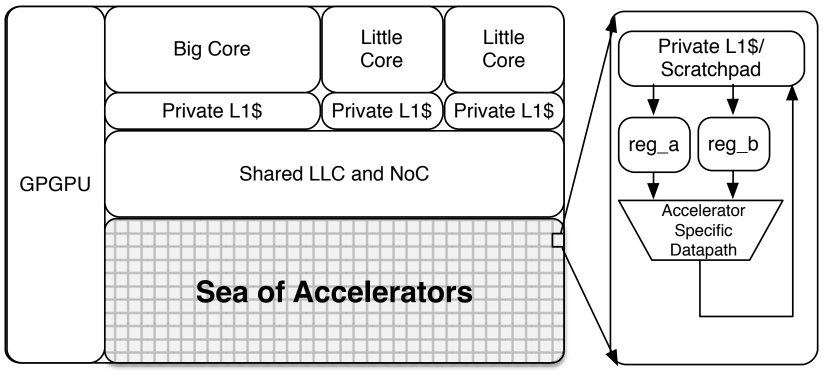


Figure 1.17: Future Heterogeneous Architecture.

1.5 Challenges in Specialized Architecture Design

It is an exciting era for specialized architectures. We are seeing accelerators being designed and deployed for machine learning, databases, networking processing, and a variety of other applications[45,128,133,144,165]. CAD vendors are introducing new high-level synthesis tools to lower the barriers of accelerator designs[1,20,28,93]. Intel's acquisition of Altera with $16 billion, Intel's largest purchase to date, marks growing excitement for integrated CPU-FPGA SoCs in servers and desktops[8]. However, architectural research in the area of specialization architectures is still in its preliminary stages. A major obstacle for such research has been the lack of an architecture-level infrastructure that analyzes and quantifies the benefits and trade-offs across different designs options.

Historically, the computer architecture community has focused on general-purpose processors, and extensive research infrastructure has been developed to support research efforts in this domain, such as workload characterization[61,71,82,107], benchmarking[31,41,151,158], and power-performance modeling[26,32,35,102,166]. However, existing accelerator design primarily relies on creating Register-Transfer Level (RTL) implementations, a tedious and time-consuming process. This slow process poses two major problems for accelerator-related research at the architectural level. First, RTL-based design flow is extremely slow. It takes hours, if not days, to generate, simulate, and synthesize RTL to get the power and performance of a single accelerator design, even with the help of high-level synthesis (HLS) tools. Such a low-level, RTL infrastructure cannot support large design space exploration to identify optimal accelerator design options. Second, architecture-level studies are generally performed in the planning stages of the design, before RTL designs have begun to take shape. This RTL-based design flow makes early-stage, architecture-level design space exploration infeasible. Hence, as the industry dives further into this era of specialization, there is a clear need for architecture-level design methodologies for early-stage accelerator studies.

1.6 Thesis Contributions

This dissertation is one of the first efforts in the computer architecture community to abstract the traditional, RTL-based design methodology to the architectural level. I have developed a series of research infrastructures, ranging from workload characterization[142], benchmarking[131], power-performance-area modeling[145,146,141], and system integration[167] to help researchers study key issues in these domains. Specifically, this research has had the following three major contributions.

1.6.1 Accelerator Workload Characterization (Chapter 3).

Specialized accelerators, especially fixed-function accelerators, are designed only for specific applications or domains of applications. Understanding *intrinsic* workload characteristics is a key, first step to deciding whether an application is amenable to acceleration. Workload characterization has been an active field of computer architecture research, though with a strong focus on general-purpose designs. Typically, hardware performance counter statistics, e.g., IPC, cache miss rates, and branch misprediction rates, are collected to characterize an application's behavior. Such an approach is useful in identifying performance bottlenecks for different applications on a specific platform, but it does not unveil applications' intrinsic characteristics, because many machine-dependent features, such as cache size and pipeline depth, can strongly affect the performance counter statistics. Hence, new workload characterization methodology is needed to help designers quickly understand the intrinsic of acceleration for applications.

To address this, we were the first to propose an ISA-independent workload characterization approach, called *WIICA*, for accelerators. WIICA leverages the existing ISA-independent nature of a compiler intermediate representation (IR). WIICA includes a compiler pass to generate a dynamic IR trace for applications of interest and profiles ISA-independent program characteristics within the broad categories of program compute, memory activity, and control flow, all of which are relevant to the design and accelerators. We also quantify the differences between ISA-dependent and -independent analysis and demonstrate that program characteristics profiled without ISA-independent analysis could be significantly biased due to microarchitecture and ISA artifacts.

1.6.2 Accelerator Pre-RTL Modeling (Chapter 4).

The increasing volume and diversity of accelerators in every generation of processors requires rolling out new designs quickly with relatively low design cost. The International Technology Roadmap for Semiconductors (ITRS) predicts hundreds to thousands of customized accelerators by 2022[18]. However, state-of-the-art accelerator research projects only contain a handful of accelerators because the cumbersome design flow inhibits computer architects from evaluating large accelerator-centric systems. Such inadequacy in infrastructure has confined the exploratory scope of accelerator research.

This led to the development of *Aladdin*, a pre-RTL, power-performance-area simulator for accelerators. Aladdin takes high-level language descriptions of algorithms as inputs and uses dynamic data dependence graphs (DDDG) as a representation of an accelerator without having to generate RTL. Starting with an unconstrained program DDDG, which corresponds to an initial representation of accelerator hardware, Aladdin applies optimizations as well as constraints to the graph to create a realistic model of accelerator activity. We rigorously validated Aladdin against RTL implementations of accelerators from both handwritten Verilog and a commercial HLS tool for a range of applications, including accelerators in Memcached[103], HARP[164], NPU[65], and a commonly used throughput-oriented benchmark suite, SHOC[58]. Our results show that Aladdin can model performance within 0.9%, power within 4.9%, and area within 6.6% compared to accelerator designs generated by traditional RTL flows. In addition, Aladdin provides these estimates over 100× faster.

1.6.3 Accelerator-System Co-Design (Chapter 5).

Specialized architectures comprising CPUs, GPUs, and accelerators are widely adopted in the mobile market and are beginning to emerge in the broad server and desktops. Typically, accelerators are often designed as standalone IP blocks that communicate with the rest of the system using a Direct Memory Access (DMA) interface. This modularity simplifies IP design and integration with the rest of the system, leaving tasks like data movement and coherency management to software device drivers. Such simplification may work fine with a handful of accelerators, but it is hard to scale with a growing number of accelerators actively requesting and operating on data at the same time. Moreover, designing each accelerator in an isolated fashion also excludes itself from taking into account possible higher-level coordination and optimization across accelerators, general-purpose cores, and shared resources such as cache hierarchies, resulting in less optimal designs.

This has motivated us to expand the Aladdin framework with other architecture-level simulators to help system architects study the interactions between accelerators and the rest of the system. Hence, we integrated Aladdin with the gem5 system simulator[32], a widely-used system simulator with configurable CPUs and memory systems. Our validation of *gem5-Aladdin* against the Xilinx Zynq SoC platform achieves less than 6.5% error. We demonstrate that co-designing accelerators with system-level considerations has two major ramifications for accelerator microarchitectures that are not yet fully understood in the literature. First, datapaths should be less aggressively parallel, which results in more balanced designs and improved energy efficiency compared to accelerators designed in isolation. Second, the choice of local memory interfaces is highly dependent on the dynamic memory characteristics of the accelerated workload, the system architecture, and the desired power/performance targets. We show that accelerator-system co-design can improve energy-delay-product by up to 7.4× and on average 2.2×.

1.7 Thesis Organization

The landscape of accelerator research has changed dramatically in the past decade. Chapter 2, therefore, takes some time to thoroughly review the efforts that both the architecture and the HLS communities have made over the years towards specialized architecture designs. Chapter 3 describes the WIICA workload characterization tool. Chapter 4 discusses the methodology and validations of the Aladdin framework for pre-RTL, power-performance-area modeling. Chapter 5 focuses on issues with accelerator-system co-design, enabled by the gem5-Aladdin infrastructure. We conclude the thesis with a discussion of future work in Chapter 6.

"A new scientific truth does not triumph by convincing its opponents and making them see the light, but rather because its opponents eventually die and a new generation grows up that is familiar with it."

Max Planck

2 Background and Related Work

This section serves as an overview of state-of-the-art accelerator research and recent advances in accelerator design flow.

2.1 Accelerator taxonomy

We characterize the space of accelerator designs with two dimensions: *coupling* and *granularity*.

Coupling defines where accelerators are deployed in the system. Today's system hierarchy typically includes a pipelined processor core with multiple levels of caches attached to the memory bus and then connected to I/O devices through the I/O bus. Conceptually, accelerators can be attached to all levels of this hierarchy, though with unique challenges and opportunities: tightly coupled accelerators require more modifications of the host processor designs, but promise lower invocation latency, while loosely coupled accelerators often incur high invocation cost, but they are freer from the design constraints of the host core. Here we discretize the degree of coupling into the following categories:

1. Accelerators that are part of the pipeline.

2. Accelerators that are attached to cache.

3. Accelerators that are attached to the memory bus.

4. Accelerators that are attached to the I/O bus.

Granularity defines what kinds of computation are offloaded to accelerators. Finer-grained accelerators are more likely to be used by a variety of applications, but they require certain changes to the software stack (i.e., ISAs, OSes, compilers, or programming languages), to allow applications to be decomposed into fine-grained regions that can be implemented by hardware accelerators when appropriate. On the other hand, coarser-grained accelerators are intended to accelerate specific functions or kernels where achieving high efficiency supersedes the need to accommodate more programs. As chips acquire more transistors than can be powered simultaneously[64], using some of those transistors to speed up an application that really matters becomes more affordable. Here we break down the computation granularity into three categories:

1. Instruction-level accelerators designed for single primitives, such as arithmetic operators (including sqrt, sin/cos).

2. Kernel-level accelerators that compose key parts of important applications. Examples include matrix multiply, stencil, and FFT.

3. Application-level accelerators that execute the entire applications such as H.264 video decoding and deep neural networks (DNN).

The Taxonomy

Table 2.1 presents the taxonomy of the state-of-the-art accelerator architecture space. Accelerators in bold are industry products, and the rest are research prototypes. Interestingly, we notice that industry products tend to reside at the top left and the bottom right corners of the table (the only exception being cryptography accelerators at the bottom left) while research projects are sprinkled almost everywhere. The reason behind this trend lies in the cost of integration.

Industry has been fairly successful in providing loosely coupled, application-level accelerators, such as GPUs or FPGA accelerators connected to the PCIe bus. Such accelerators provide off-the-shelf solutions that minimally interfere with the hardware design of general-purpose cores or the existing CPU software stack. On the hardware side, today's

	Part of the Pipeline	Attached to Cache	Attached to the Memory Bus	Attached to the I/O Bus
Instruction-Level	**FPU,** **SIMD,** DySER[74],	Hwacha[100,122,161], CHARM[54,51],		
Kernel-Level	NPU[65], 10x10[47], Convolution Engine[129], H.264[78],	DANA[62], SNNAP[117], C-Cores[159],	Database[40], Q100[165], LINQits[48], AccStore[110],	
Application-Level	**x86 AES[11], Oracle/Cavium Crypto Acc[15,90],**	Key-Value Stores[112], Memcached[103],	Sonic3D[136], DianNao[43,44], HARP[164], **TI OMAP5[19], IBM PowerEN[94], IBM POWER7+[33],**	**GPU, Catapult[128], IBM Power8 CAPI Acc[5],**

Table 2.1: Accelerator taxonomy.

commercial accelerators require minimal changes in general-purpose core designs; accelerators either reside on a separate chip, as in the case of GPUs or FPGA accelerators, or they are plugged in as standalone IP blocks, as in the case of today's SoCs. On the software side, the software stack is also more or less intact: the entire workload is offloaded to accelerators, and no additional management is needed.

Industry has also been good at providing tightly coupled, instruction-level accelerators, such as FPU and SIMD units, a category located at the opposite corner of the table from loosely coupled, application-level accelerators. The wide adoption of accelerators in this category also stems from the relative ease with which they can be integrated. Of course, certain modifications to ISAs and compilers are needed to allow applications to leverage accelerators inside the pipeline, such as auto vectorization for vector units, and hardware designers need to balance the new execution units with the rest of the pipeline. However, the changes are well-contained inside the pipeline, with little impact on the memory system.

The accelerator space that requires more attention in the next decade lies in cache-attached, kernel-level accelerators. These are the areas where we see little industry presence yet and mostly early work from research projects. In terms of coupling, tightly coupled cache-attached accelerators free programmers from worrying about low-level data movement between accelerators and cores, as is the case of loosely coupled accelerators with DMA, but they need to cope with non-uniform memory latency, virtual memory, and cache coherence on their own[147]. In terms of granularity, kernel-level accelerators attempt to strike a balance between speedups and programmability by carving out significant chunks of computation, while providing reuse across applications. To decompose applications into kernel-level accelerators, detailed workload characterization is needed to understand the speedup potential for different kernels.

Here we discuss some recent projects on hardware accelerators in each category of the taxonomy. The list of examples is not intended to be exhaustive.

2.1.1 Accelerators that are part of the pipeline.

One philosophy of integrating accelerators holds that if the accelerator is important enough, it should be put inside the pipeline as an execution unit. A classic example is the floating point unit (FPU). The advantage of integrating accelerators into the pipeline is that the accelerator design does not need to worry about the accelerator's interaction with the rest of the system: accelerators are just new functional units inside the pipelines, and they can leverage a core's load/store units and TLB for memory accesses. On the other hand, the performance of accelerators in this category can easily be limited by constraints of the cores, e.g., the bandwidth of the register files. These benefits and limitations are true for all the pipeline-attached accelerators. In this section, we have a representative accelerator design for each granularity in this category.

Instruction-Level

We start with instruction-level accelerators that are part of the pipeline.

FPU and SIMD. In early Intel chips, there were no floating point units. If the program required floating-point computing, programmers needed to emulate those computations in software, which was very slow. Eventually, with the increasing need for floating-point capability, floating-point processing was built into hardware. However, the first floating point unit was shipped as a co-processor on another chip (8087), which users could attach to the main CPU chip (8086) if desired. Not until the 80486 did Intel start integrating the floating-point unit with the CPU. Now the floating-point unit is a default component in a most processor core pipelines. Similar evolution also happened with Single Instruction Multiple Data (SIMD) units that speed up data-parallel computation.

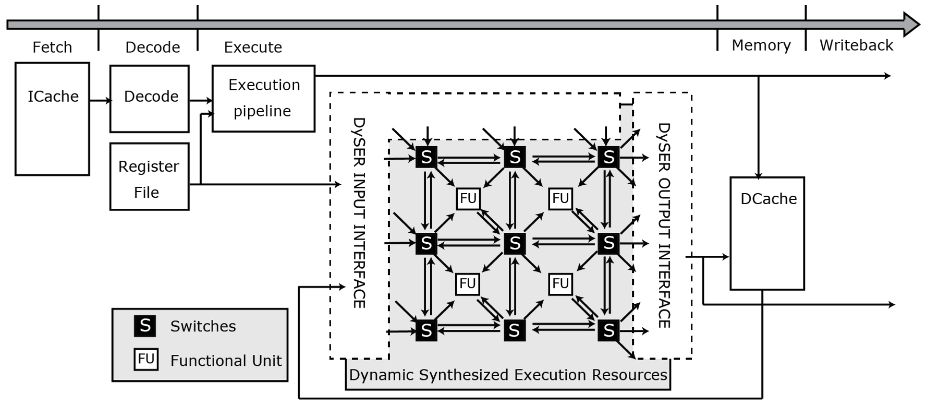


Figure 2.1: DySER pipeline[74].

DySER. Dynamically Specialized Execution Resource (DySER) is a hardware-compiler co-design approach to dynamically wire a mesh of functional units for different phases of program execution. Each phase is roughly tens to hundreds of instructions. The key idea of DySER is to speed up commonly-used computation paths without incurring repeated fetch, decode, and register access costs. DySER proposed an accelerator architecture integrated into the host processor's pipeline, shown in Figure 2.1, as a special execution unit. The DySER accelerator is composed of a heterogeneous array of functional units (FUs) connected with a mesh network. The granularity of the functional units in DySER is similar to the complexity of instructions in CPUs. Examples of FUs include integer ALU, multiply, and divide, as well as floating-point add/subtract, multiply, and divide[74].

Kernel-Level

Compared to instruction-level accelerators, kernel-level accelerators implement large, more semantically rich functions, e.g., multiple basic blocks, of computation.

NPU. The neural processing unit (NPU) is a low-power, neural networks accelerators targeted at emerging approximate applications that can tolerate inexact computation in

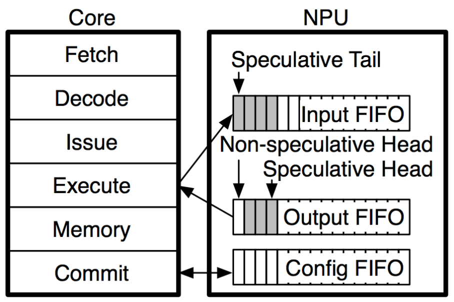


Figure 2.2: Neural processing unit[65].

substantial portions of their executions[65]. The NPU is tightly coupled to the processor pipeline, shown in Figure 2.2, to accelerate small kernels, e.g., fft and sobel edge detector, in approximate applications. When a programmer writes their code for NPU, they explicitly annotate functions that are amenable to approximate execution. During compilation time, a NPU compatible compiler trains a neural network for the candidate region based on input-output training data, and generates codes that configures the NPU before its invocation. The NPU configuration and invocation is done through ISA extensions that are added to the core.

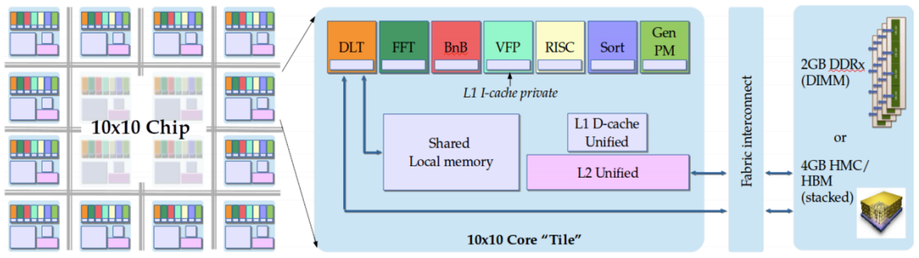


Figure 2.3: An example of 10x10 architecture[47].

10x10. The 10x10 project envisions a customized, but general-purpose architecture, that exploits the benefits of customization for energy efficiency and performance, but maintains programmability and parallel scalability[47]. The goal of the project is to identify 10 most important kernels and then accelerate each kernel to be 10× more energy efficient and 10× faster[46]. The application coverage is achieved by optimizing ten distinct, but commonly used kernels. An example of the 10x10 architecture is shown in Figure 2.3. It has six kernel-level accelerators (micro-engines), including FFT, sort, and pattern matching (GenPM), and one RISC core. Each of the micro-engines is basically a specialized core with a customized functional units for the target kernels. New instructions are added to the ISA to invoke these specialized functional units.

Convolution Engine. Convolution engines (CEs) are specialized for convolution-like dataflow kernels that are common in computational photography, image and video processing applications[129]. CE is developed as a specialized functional unit to Tensilica's extensible RISC cores[86]. Specific CE instructions are added to the ISA. The host RISC core decodes instructions in its instruction fetch unit and routes the appropriate control signals to CE if a CE instruction is encountered. The host core is also responsible for memory address generation, but the data is sent/returned directly from the internal register files in CE.

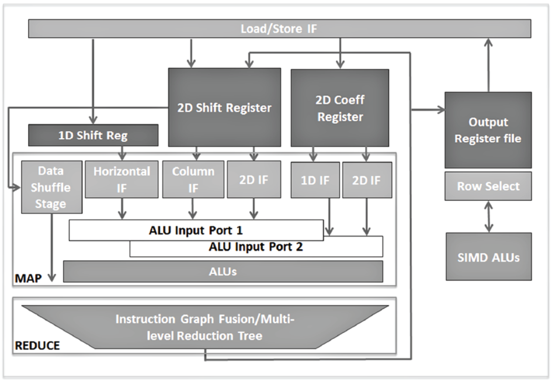


Figure 2.4: Convolution engine system overview[129].

H.264 Accelerators. A recent project from Stanford asked an interesting question: what are the sources of inefficiency in general-purpose processors[78]. The authors started from a 720p H.264 encoder running on a general-purpose processor, where the corresponding ASIC implementation is 500× more efficient than the general-purpose baseline. The paper then explores methods to eliminate general-purpose overheads by gradually transforming the CPU into a specialized system for H.264 encoding. Customized functional units, e.g., SIMD and fused operations, and customized storage, e.g., shift registers, are added to speedup important kernels inside the H.264 encoder. Similar to the Convolution Engine approach, this work also leverages the capabilities of Tensilica processors to add customized instructions into the host ISA. An example of the Tensilica processor is shown in Figure 2.5.

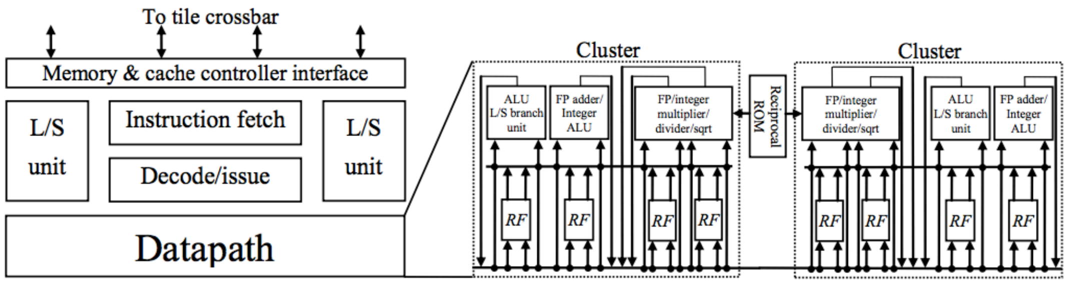


Figure 2.5: An example of the Tensilica processor pipeline[149].

Application-Level

The most common application-level accelerators that are being integrated into the pipeline are cryptography accelerators[11,15,90]. Encryption/decryption algorithms are usually quite suitable for acceleration because of their high computational requirements and mature standardization. Implementations of the AES instruction set integrate the encryption/decryption accelerators into the processor pipeline and provide nearly an order of magnitude improvement in AES throughput. The Advanced Encryption Standard Instruction Set (AES-NI) is an extension of the x86 ISA for encryption and decryption using AES[11]. Oracle's SPARC and Cavium OCTEON II also include cryptography accelerators inside the core to accelerate security applications[15,90].

2.1.2 Accelerators that are attached to cache

Both pipeline- and cache-coupled accelerators have an address space unified with the host cores. However, an important distinction between them is that cache-coupled accelerators need to access the shared-coherent cache hierarchy without assistance from the host core's TLB or load-store unit. Typically, accelerators in this category also need to implement their own units for data movement and address translation.

Instruction-Level

Hwacha. The Hwacha project focuses on vector processing for data-parallel applications[100,122,161]. Figure 2.6 shows an example of a Hwacha accelerator. The Rocket scalar core in Figure 2.6 is a RISC-V core from UC Berkeley. A Hwacha accelerator is a vector co-processor that has its own instruction cache, but shares a data cache with the host

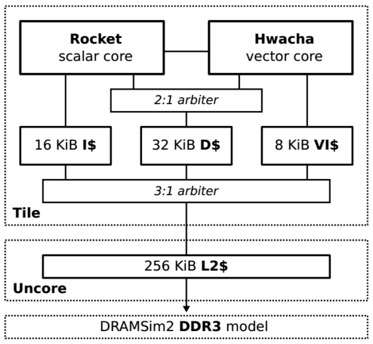


Figure 2.6: An example of Hwacha vector accelerators[122].

core. As a good example of cache-coupled accelerators, a Hwacha accelerator has its own internal TLB, managed by the OS, for address translation and a load-store unit for cache access.

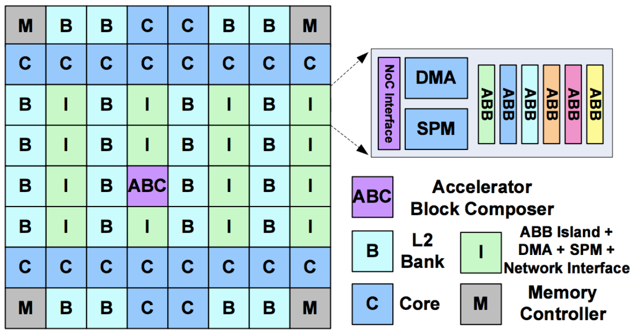

NoC Interface
DMA
SPM
ABB
ABC
Accelerator Block Composer
B
L2 Bank
I
ABB Island + DMA + SPM + Network Interface
C
Core
M
Memory Controller

Figure 2.7: CHARM architecture[54].

CHARM and CAMEL. The Composable Heterogeneous Accelerator-Rich Microprocessor (CHARM) project focuses on composable accelerator blocks at instruction granularity that can be dynamically composed for different workloads[54]. Figure 2.7 shows an overview of the CHARM architecture. CHARM is a heterogeneous architecture with cores, L2 caches, memory controllers, accelerator building block (ABB) islands (I), and an accelerator block composer (ABC) that dynamically connects different ABBs together for different functionalities. Inside each ABB is a dedicated scratchpad memory (SPM), a DMA engine, a NoC interface, and a set of fine-grained accelerators, such as reciprocal, square-root, polynomial-16, and divide. CHARM supports a unified address space across the entire architecture, where a small, internal TLB is included in the DMA engine of each ABB. In the event of a TLB miss, the TLB request is forwarded to a shared TLB inside the accelerator block composer. If it also misses the shared TLB, the request is forwarded to the host core. CAMEL extends the CHARM architecture from ASIC to FPGA for better reconfigurability[51].

Kernel-Level

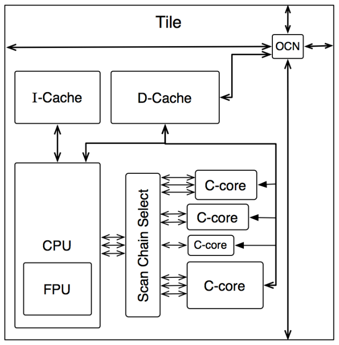


Figure 2.8: A C-core-enabled system[159].

C-core, ECOcore, and QsCore. A series of efforts from the University of California, San Diego, focuses on energy-efficient, but not necessarily better performing, accelerators called Conservation Cores (C-cores)[159,135,160]. C-cores efficiently execute hot regions of specific applications that represent significant fractions of the target system's workload. An example of C-core architecture is shown in Figure 2.8. The entire C-core tool chain starts with extracting the most frequently used code regions and synthesizing them into C-core hardware. In order to generate code for the C-core augmented processor, it extends a compiler framework with a combination of OpenIMPACT and GCC with knowledge of existing C-core on chip. The compiler uses a matching algorithm to find similarities between the input code and the C-core specifications. In the case of matching, the compiler generates a C-core-enabled binary that makes use of the C-core. The C-core accelerator shares the L1 cache of the host core, though the host CPU and C-cores do not simultaneously access the cache. The design assumes a coherent cache interface to enable communication between the C-core accelerator and the host core, but it does not mention how address translation is supported.

SNNAP. SNNAP is an FPGA prototype for neural network accelerators[117]. Instead of adding a neural network accelerator into a processor's pipeline, as in the case of NPU[65], SNNAP implements accelerators on an on-chip FPGAs, avoiding changes to the processor's ISA and microarchitecture. To program a SNNAP accelerator, application programmers can either use a high-level, compiler-assisted mechanism that automatically transforms regions of approximate code to offload them to SNNAP accelerators, or a low-level, explicit interface that can batch multiple invocations together for pipelined processing. In the runtime, when a SNNAP-compatible program starts, it first configures the SNNAP accelerator with its topology and weights using the General Purpose I/Os (GPIOs) interface. Then, the program sends inputs through ARM Accelerator Coherency Port (ACP). The host processor then uses the ARMv7 `SEV/WFE` signaling instructions to invoke the SNNAP accelerator. The accelerator writes outputs back to the processor's cache via the ACP interface, and, when finished, signals the processor to wake up.

Application-Level

Memcached. In-memory, key-value stores are an important component of modern data center services[103,112]. `Memcached` is implemented using a hash table, with a unique key that indexes the stored data. Figure 2.10 depicts a recent design of a `memcached` accelerator, called Thin Servers with Smart Pipes (TSSP). TSSP is designed for cost-effective, high-performance `memcached` deployment. It couples an embedded-class low-power core to a `memcached` accelerator that can process GET requests entirely in hardware. A system MMU translates virtual addresses to physical addresses that can be

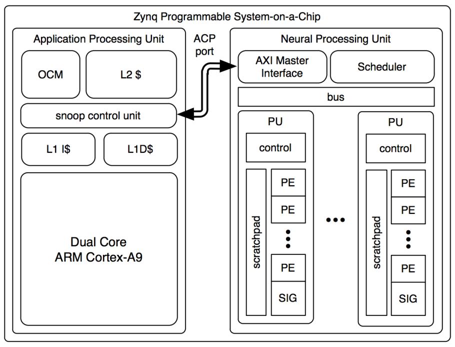


Figure 2.9: SNNAP[117].

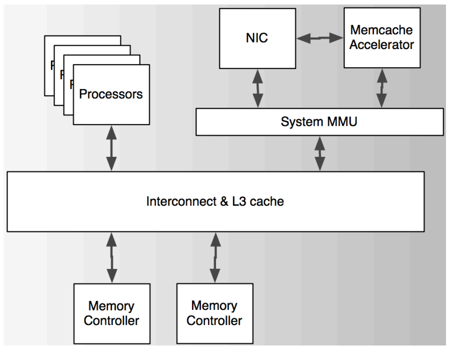


Figure 2.10: The overall Memcached architecture[103].

shared between the accelerators and the cores.

2.1.3 Accelerators that are attached to the memory bus

Accelerators that are attached to the memory bus are usually coarser-grained accelerators since the invocation and offloading cost is more significant. Accelerators proposed in this category usually do not have much interaction with the host cores: they can access their own physical memory directly without worrying about address translation and coherence.

Kernel-Level

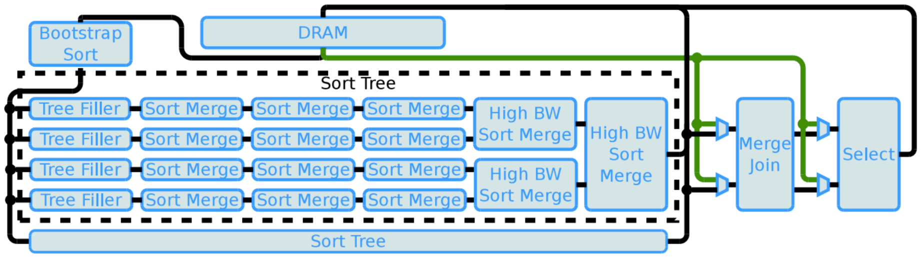


Figure 2.11: Hardware acceleration of database operations system overview[40].

Database Acceleration. Database processing is an emerging area for acceleration. Its high memory bandwidth requirement makes it a perfect candidate for near-memory accelerators. Casper and Olukotun present an FPGA prototype using hardware to accelerate three important in-memory database primitives: selection, merge join, and sorting[40], all of which are connected directly to memory, as shown in Figure 2.11.

Q100. The Q100 is a Database Processing Unit (DPU) that can efficiently handle data-analytic applications[165]. It has a collection of heterogeneous fixed-function ASIC tiles, listed in Figure 2.12. Each of the tiles implements a database relational operator, such as a joiner or sorter, that resemble a common SQL operator. These tiles communicate with each other through an on-chip interconnect and load (store) data from (to) off-chip memory.

LINQits. The LINQits framework provides a pre-tuned accelerator template to accelerate a domain-specific query language, i.e., Language-Integrated Query (LINQ)[48].

	Area		Power		Critical Path
Tile	mm^2	**% Xeon** [a]	**mW**	**% Xeon**	**ns**
Aggregator	0.029	0.07%	7.1	0.14%	1.95
ALU	0.091	0.21%	12.0	0.24%	0.29
BoolGen	0.003	0.01%	0.2	<0.01%	0.41
ColFilter	0.001	<0.01%	0.1	<0.01%	0.23
Joiner	0.016	0.04%	2.6	0.05%	0.51
Partitioner	0.942	2.20%	28.8	0.58%	***3.17
Sorter	0.188	0.44%	39.4	0.79%	2.48
Append	0.011	0.03%	5.4	0.11%	0.37
ColSelect	0.049	0.11%	8.0	0.16%	0.35
Concat	0.003	0.01%	1.2	0.02%	0.28
Stitch	0.011	0.03%	5.4	0.11%	0.37

Figure 2.12: Area/power/delay characteristics of Q100 tiles compared to a Xeon core[165].

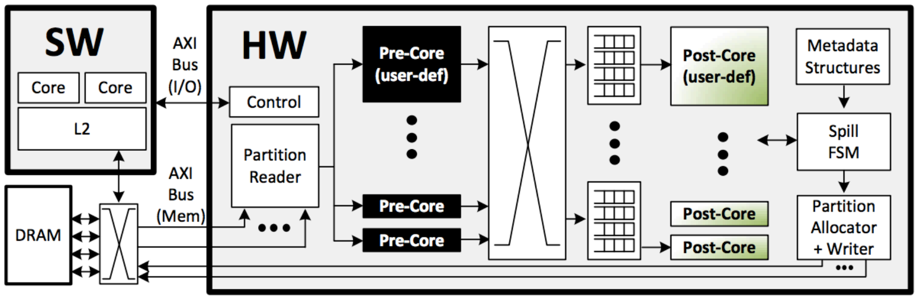


Figure 2.13: LINQits hardware overview[48].

LINQ supports a set of operators based on functional programming patterns and has been used to implement a variety of applications. Unlike other query languages, LINQ operators accept user-defined operators through a functional declaration as part of its processing. LINQits builds LINQ operators onto an SoC accelerator framework and provides a reconfigurable template to express user-defined functions. Figure 2.13 shows an overview of the LINQits framework. Data is streamed from main memory through the partition reader, which is a specialized DMA engine responsible for reading bulk data.

Accelerator Store. The Accelerator Store project proposed a shared memory framework for many-accelerator architectures. An accelerator characterization of a diverse pool of accelerators reveals that each accelerator contains significant amounts of SRAM memory (40% to 90% of the total area)[110]. The characterization also shows that large private SRAM memories embedded within accelerators tend to have modest bandwidth and latency requirements. Figure 2.14 gives an example of accelerator store design. An accelerator store is a pool of SRAM memories connected through the memory bus. The pool can be shared by multiple accelerators. The handle table provides virtual

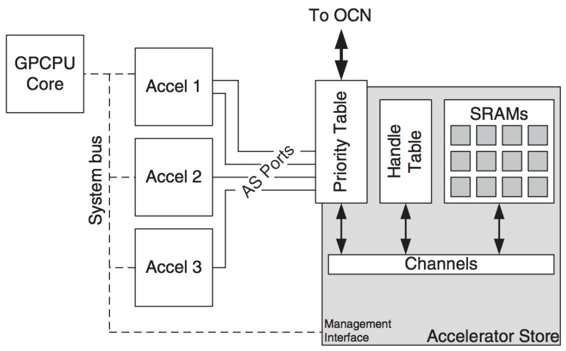


Figure 2.14: Accelerator store design[110].

memory support between them. An accelerator store can reduce the amount of on-chip memory for many-accelerator architectures with low overhead in terms of performance and energy.

Application-Level

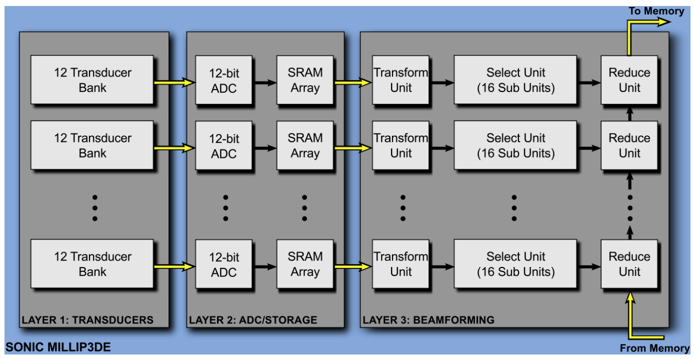


Figure 2.15: The Sonic Millip3De Hardware Overview[136].

Sonic3D. Figure 2.15 gives an overview of a standalone ASIC accelerator for 3D ultrasound beam formation—the most computationally intensive aspect of image formation[136]. The accelerator reads the input image from memory and writes the resulting updated image back to memory.

DianNao. Machine learning is another emerging area where specialized accelerators could significantly improve performance and energy efficiency. One example of on-going

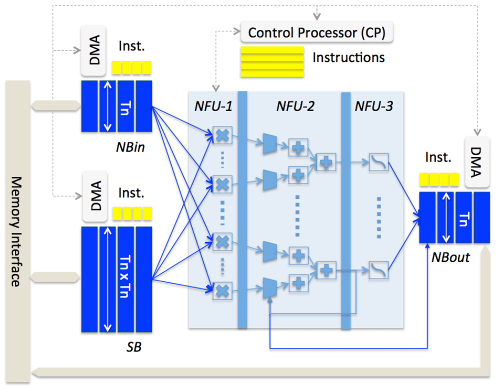


Figure 2.16: Diannao Accelerator[43].

projects in this area is the Diannao project[43,44]. Diannao is an accelerator design for neural network algorithms, i.e., convolution neural networks (CNNs) and deep neural networks (DNNs), shown in Figure 2.16. DMA engines transfer data between accelerator buffers and memory.

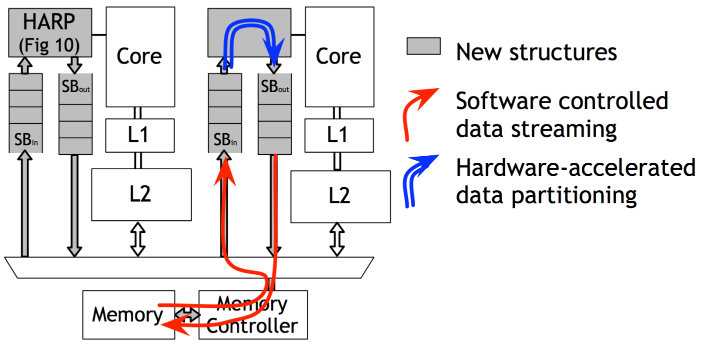


Figure 2.17: A 2-core system w/ HARP integration[164].

HARP. HARP is a hardware accelerator for range partitioning, which is central to modern database systems, especially for big-data analytics[164]. Figure 2.17 shows a block diagram of the major components in a system with HARP accelerators. The HARP accelerator is connected to the memory bus through two buffers; software moves data in and out of the buffers. A set of instructions is added to the host ISA for data movement orchestration between memory and HARP buffers. With data streaming in, the range partitioning is accelerated in the hardware accelerator.

2.1.4 Accelerators that are attached to the I/O bus

Here we move beyond the chip boundary: from on-chip accelerators to off-chip accelerators. A good example of programmable accelerators in this category is off-chip GPU with its own DRAM. Accelerators in this category are loosely coupled with the host without much fine-grained communication. Usually significant speedup from the accelerator is expected to offset the communication cost.

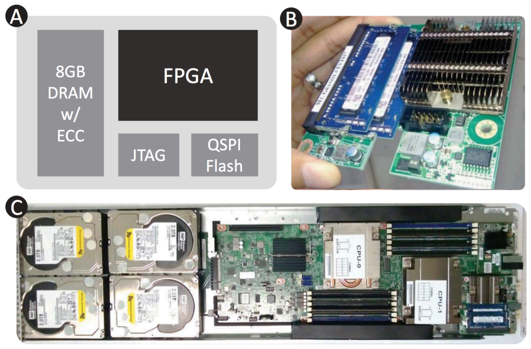


Figure 2.18: (a) Catapult FPGA block diagram. (b) Manufactured board. (c) The server that hosts the FPGA board.[128].

Catapult. Catapult is a prototype FPGA-based accelerator for the Bing web search engine from Microsoft[128]. Catapult FPGAs, each with their own DRAM, are embedded into a half-rack of 48 machines. FPGAs are directly connected to each other in a 6x8 two-dimensional torus through general-purpose I/Os. Catapult is used to accelerate part of Microsoft Bing's ranking algorithm. When a server needs to rank a document, the software converts the document into a Catapult-compatible format and injects the document into its local FPGA. The FPGA pipeline computes the score for the document and sends the result back to the requesting server. Catapult was deployed with 1,632 servers running with mirrored Bing search traffic. Compared to a software-only implementation, Catapult achieves a 95% improvement in throughput with an equivalent latency distribution. On the other hand, it reduces the tail latency by 29% at the same throughput.

2.2 Standard RTL Design Flow

Despite the increasing popularity of hardware accelerators, the standard accelerator design flow is still quite low-level and requires the use of complex and time-consuming electronic design automation (EDA) tools. Figure 2.19 illustrates the design flow for ASIC accelerators. It starts with a high-level description of an algorithm, then designers either manually implement the algorithm in Register-Transfer Level (RTL) using Verilog or VHDL or use high-level synthesis (HLS) tools, to compile the high-level programs to RTL. Hardware description languages such as Verilog and VHDL date back to the 1980's. They allow hardware designers to describe circuits using low-level building blocks, such as multipliers, registers, and multiplexers. Functional verification tests whether the resulting RTL design agrees with specifications. When all the blocks are implemented and verified, designers use commercial logic synthesis tools to map their designs to the gate level. Tools such as Synopsys Design Compiler take as input RTL in Verilog or VHDL, target technology libraries, and constraints, and they produce a gate-level netlist. This netlist is then transformed by place-and-route tools, such as Cadence SoC Encounter into the physical circuit layout. The whole process illustrated in Figure 2.19 is iterative, requiring a lot of tuning and refining at each stage.

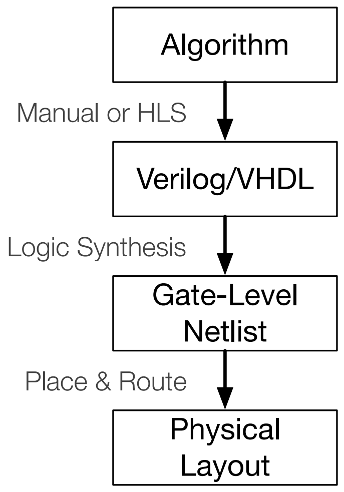


Figure 2.19: Synthesis flow.

2.3 High-Level Synthesis

Today's hardware designs are predominantly programmed using hardware description languages, such as Verilog and VHDL. Producing efficient designs using such inherently low-level languages requires a lot of expertise. It is time-consuming even for experienced hardware designers. Such a slow design process will not scale with tight SoC design cycles and the increasing use of hardware accelerators. Moreover, the low-level programming model also discourages application programmers, who have little hardware design knowledge, from converting applications into hardware. In addition, the history of software development in the past half century has demonstrated the value of higher abstraction levels to tackle growing complexity. For example, CPU programming benefits from well-established abstraction interfaces and advanced compilers, which free programmers

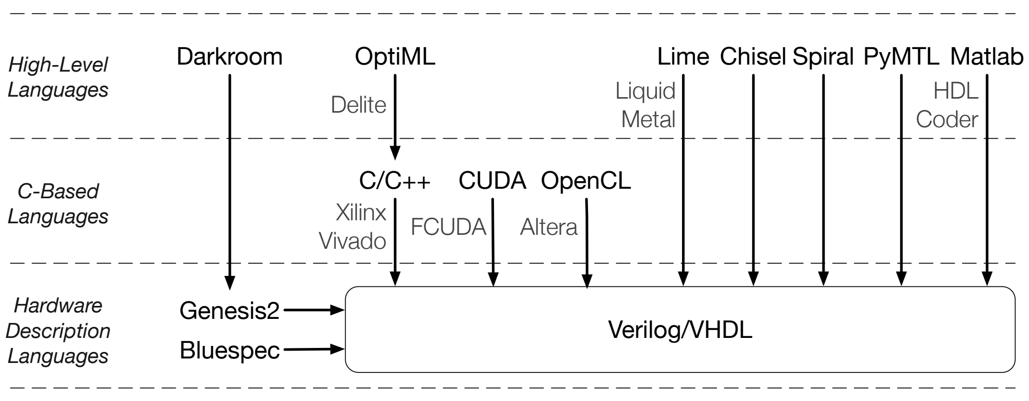


Figure 2.20: High-level synthesis landscape.

from low-level details, increase their productivity, and reduce the likelihood of bugs.

To raise the level of abstraction in hardware designs, high-level synthesis tools that convert a high-level algorithm description into low-level RTL code are getting more attention in both academia and industry. The benefits of high-level synthesis include:

1. Better design. Higher abstraction levels free designers from worrying about low-level implementation details. With the help of a fast automation flow, designers can spend more time exploring alternative designs in terms of power, performance, and cost, potentially leading to better design choices.

2. Lower design cost. SoC companies have tight design cycles. Greater automation in the design flow shortens design time and reduces expensive human involvement.

3. Greater accessibility for application designers. The future of specialization requires a lot of hardware-software co-design. Higher abstraction levels in hardware design could empower application programmers to more easily evaluate high-level algorithms in hardware.

Figure 2.20 summarizes the state-of-the-art high-level synthesis frameworks in both the commercial and research spaces. We divide high-level synthesis frameworks into three categories: hardware-description-language-based flow, e.g., Bluespec[120] and Genesis2[139]; C-like-language-based flow, e.g., Xilinx Vivado[20], FCUDA[123], Altera SDK for OpenCL[1]; and high-level-language-based, e.g., Darkroom[81], OptiML/Delite[73,154,93,153], Lime/Liquid Metal[25,24], Chisel[28], Spiral[57,113], PyMTL[105], and Matlab HDL Coder[13].

The rest of this chapter aims to provide a pointer to some of the efforts in state-of-the-art tools. Interested readers can refer to the original papers/manuals to find more details.

Bluespec SystemVerilog   Bluespec SystemVerilog (BSV) is one of the early efforts to provide a higher-level hardware description language than Verilog/VHDL. Based on SystemVerilog syntax, Bluespec supports a higher level of abstraction than Verilog/VHDL in both behavioral and structural descriptions. Inspired by functional programming languages, such as Haskell, Bluespec supports more expressive types, overloading, encapsulation, and flexible parametrization to enable code reuse[120]. Programs written in Bluespec are compiled using the Bluespec compiler to generate corresponding RTL descriptions.

Genesis2   Genesis2 is designed to create domain-specific hardware generators that encapsulate designer knowledge and only expose high-level application parameters. When running a generator, a user provides a set of parameters and constraints. From these, the generator produces the desired module. Instead of introducing a new hardware description language, Genesis2 extends SystemVerilog to exploit its verification support, and it uses Perl to provide flexible parametrization[139]. Darkroom[81] is an example of using Genesis2 to build image-processing pipelines in hardware.

Xilinx Vivado   Almost all CAD vendors have developed high-level synthesis tools, e.g., Cadence C-to-Slicing Compiler, Synopsys Synphony C Compiler, Mentor Graphics Catapult C, and Xilinx Vivado. Originally AutoESL, Vivado takes programs written in C/C++/SystemC as well as user-defined directives (similar to pragmas) and generates RTL. A subset of the C language is supported; features, such as recursive functions and dynamic memory allocation are prohibited. As with other synthesis flows, the quality of Vivado-generated RTL designs is highly dependent on the input C code quality.

Delite   There is increasing interest in domain-specific languages (DSL) for hardware compilation. DSLs can incorporate high-level, domain-specific features, and they can enforce restrictions that are not valid in general-purpose programming, making compiler analysis less conservative. Delite is a compiler framework that facilitates DSL development[154]. Delite simplifies DSL construction by providing common reusable components, such as parallel patterns, optimizations, and code generations. Delite-generated DSLs, e.g., OptiML[153], are embedded in Scala, a general-purpose functional programming language[17], and can be compiled to multiple languages, such as C++, CUDA, and OpenCL. Delite-generated C code, optimized with domain-specific knowledge conveyed in DSLs, can be used as input to C-based high-level synthesis tools, such as Xilinx Vivado, to generate better RTL implementations[73,93].

LIME The Liquid Metal project is a compiler and runtime system for heterogeneous architectures with a single and unified programming language[24]. This new programming language, called Lime, is intended to be executable across a broad range of architectures, from FPGAs to CPUs[25]. Based on Java, Lime has machine-independent semantics and a dynamic execution model, with extensions for parallelism, isolation and data flow. The compiler frontend performs optimization and compiles source code written in Lime to Java bytecode. The backend generates code for GPUs and FPGAs. The runtime dynamically picks the best implementation of a task based on available resources.

CHISEL Chisel is a hardware construction language embedded in Scala[28]. Chisel includes a set of Scala libraries that define hardware datatypes and a set of routines to compile source code into either a cycle-accurate C++ simulator or a Verilog implementation. Examples of hardware built using Chisel include the RISC-V Rocket core[16], a key-value store accelerator[112], and the Hwacha vector accelerator[100].

SPIRAL Spiral is a domain-specific hardware/library generator for signal processing[118,57,113]. In Spiral, a signal processing algorithm is first described as a set of formulas expressed in Spiral's signal processing language (SPL). Spiral recursively applies different transformations and optimizations to generate an optimal design based on program input and available hardware resources. Then the optimized representation is compiled into C or Verilog.

PYMTL PyMTL aims to support computer architecture research by providing a vertically unified design environment for function-level, cycle-level, and RTL simulation[105]. Based-on Python 2.7, PyMTL allows designers to build architectural simulators at the function, cycle, or RTL level using a single language interface. For RTL simulation, PyMTL includes an RTL translator that translates the Python description of hardware into Verilog, which then is ported to the standard EDA design flow to produce power, area, and performance estimates.

2.4 Putting it Together

We are seeing a vibrant research community in the fields of computer architecture and VLSI-CAD that embrace the era of specialization by proposing novel accelerator architectures for a wide range of applications and new tools to ease the RTL generation process. However, when it comes to the implementation and evaluation parts of the work, we see that most of the architectural research has to use the slow and tedious RTL simulation and synthesis process. Although with the recent advance in high-level synthesis flow that has made it easier to generate RTL designs, it still takes hours, if not days, to simulate and synthesis the RTL designs in order to get accurate power and performance estimates. Such slow evaluation flow has confined the exploratory scope of possible design space exploration for heterogeneous architecture. The following three chapters will discuss three major accelerator research tools that I have built during my PhD on workload characterization, power-performance-area modeling, and system integration, all of which aim to abstract the accelerator design process to the architectural level.

"If you know the enemy and know yourself, you need not fear the result of a hundred battles. If you know yourself but not the enemy, for every victory gained you will also suffer a defeat. If you know neither the enemy nor yourself, you will succumb in every battle."

Sun Tzu, *The Art of War*.

3 WIICA: ISA-Independent Workload Characterization for Accelerators

Accelerators are intrinsically tailored to applications, and workload characterization plays a large role in developing these architectures. Tuning an architecture towards a workload requirement demands a comprehensive understanding of the intrinsic characteristics of the workload. This chapter is about workload characterization for accelerator design. We will introduce WIICA, an Instruction Set Architecture (ISA)-independent workload characterization tool.

3.1 Introduction

Workload characterization for general-purpose architectures is commonly done by profiling benchmarks on current generation microprocessors using hardware performance counters. Typical program characteristics are machine instruction mix, IPC, cache miss rates, and branch misprediction rates. This approach is limited because machine-dependent features such as cache size and pipeline depth strongly affect the characterization of the workload. To overcome this problem, *microarchitecture-independent* workload characterization can be employed by profiling instruction traces to collect information such as working set sizes, register traffic, memory locality, and branch predictability[82]. Although this approach removes the effects of microarchitecture-dependent features, some of these analyses depend on the particular ISA used to collect the trace. Each ISA has different characteristics and constraints that impact the representation of the workload. As architectural specialization grows in importance, *ISA-independent* workload characterization will become essential for understanding intrinsic workload behavior, which will in turn allow designers to consider a wide range of alternative architectures.

To fully expose the microarchitecture- and ISA-independent workload characteristics for specialized architectures, we propose to analyze benchmarks using ISA-independent characteristics that capture inherent program behavior. To perform this analysis, we leverage the existing ISA-independent nature of a compiler intermediate representation (IR). We use a JIT compiler to trace workloads using this IR and compare program characterization within the broad categories of program compute, memory activity, and control flow. In particular, we study program characteristics that are highly relevant to the design of specialized architectures. Within each category, we analyze and discuss the differences between ISA-independent and ISA-specific analysis. Finally, we demonstrate cases where the ISA-independent characterization can help designers categorize workloads into different specialization approaches. In particular, this chapter makes the following contributions:

1. We compare ISA-dependent characterization with ISA-independent characterization. To the best of our knowledge, this is the first such ISA-independent workload characterization study. We show that ISA-dependent results can be misleading. In particular, the memory behavior of the workloads, which is critical for many forms of architectural specialization, will be biased significantly due to the register spilling effect intrinsic to conventional ISAs.

2. We present a taxonomy to characterize the potential for architectural specialization using ISA-independent characteristics. We categorize a workload's ISA-independent characteristics into program compute, memory activity, and control flow, each of which corresponds to an important component of specialized architectures.

3. We present workload characterization of SPEC CPU benchmarks using ISA-independent characteristics, and we demonstrate that a truly intrinsic workload characterization allows accelerator designers to quickly identify opportunities for specialization.

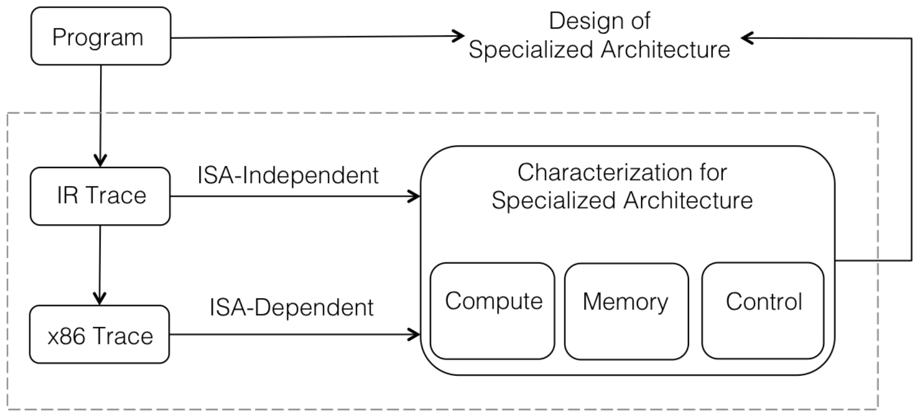

Program
Design of Specialized Architecture
IR Trace
ISA-Independent
Characterization for Specialized Architecture
x86 Trace
ISA-Dependent
Compute
Memory
Control

Figure 3.1: WIICA Overview.

3.2 Motivation

Accelerators are unburdened by the requirements of legacy ISAs, and much of the efficiency gained by using accelerators can be attributed to hardware specialization of the datapath, memory, and program control. ISA-independent analysis is attractive for such architectures because it avoids artificial constraints imposed by details of a specific ISA. These constraints affect the behavior of programs because compilers for conventional ISAs must generate binaries that meet the specification of the instruction set semantics. This subsection discusses the effects of three major kinds of ISA constraints: the overhead of stack operations caused by register spilling, ISA-specific complex operators, and calling conventions.

3.2.1 Stack Overhead.

The set of registers defined by an instruction set architecture is necessarily smaller than or equal to the set of physical registers in a machine. Most high-level language code uses variables liberally, without regard for the number of registers in a particular target system. To fit the large number of variables into the ISA-defined register set, compilers must perform register allocation to map program variables to registers. When there are more live variables needed than available architectural registers, the compiler spills some of the variables onto the stack. Load and store operations inserted move spilled values into and out of machine registers for computation. These stack memory operations can be expensive from a run-time performance point of view. When characterizing workloads for specialized architectures that do not have a fixed or known ISA, such stack accesses add irrelevant load/store operations that distort memory utilization.

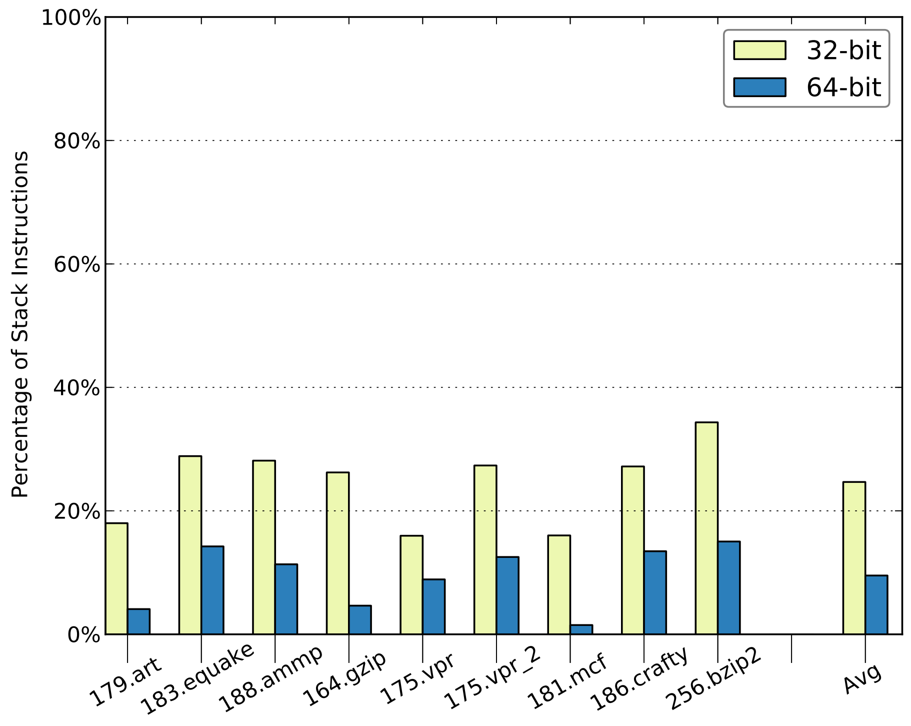


Figure 3.2: The percentage of stack instructions of total dynamic instructions for 32-bit and 64-bit x86 binaries.

To demonstrate the effect of stack operations, we compare 32-bit and 64-bit x86 binaries generated by LLVM's Clang compiler for a set of SPEC CPU benchmarks. One of the major differences between the 32- and 64-bit x86 ISAs is that the 64-bit x86 ISA has eight more general-purpose registers. Figure 3.2 plots the percentage of dynamic instructions that access the stack for 32-bit and 64-bit implementations of the benchmark set. Note that for each benchmark, the 32-bit version executes a much higher percentage of stack instructions than the 64-bit version. Additional general-purpose registers allow more variables to stay in registers, so less spilling to memory is required.

The stack overhead also applies to RISC ISAs. Lee et al. characterize stack access frequency using the Alpha ISA to propose a mechanism for separating stack from heap accesses[99]. For the same SPEC CPU2000 workloads, they find a percentage of stack operations (24%) that is similar that in Figure 3.2 for 32-bit x86.

3.2.2 Complex Operations.

Two classes of instructions can be categorized as complex operations: vector instructions and compute or branch instructions with memory operands. Both kinds of operations can be split into multiple simpler primitives. CISC ISAs. For example, x86 contain complex operations including vector instructions, such as SSE and instructions that support memory operands. However, complex operations can exist even in RISC ISAs. For example, POWER and ARM include complex operations such as predicate instructions, string instructions, and vector extensions.

When we perform workload characterization for specialized architectures, it is easier and cleaner to start from simple primitives and explore aggregation possibilities rather than to start from a more complex version of code resulting from another category of optimization.

Figure 3.3 quantifies the amount of complex operations in x86. In this categorization, an instruction is classified as a complex operation if it is either a vector instruction (SSE) or a compute or branch instruction with a memory operand. The top three categories in Figure 3.3 are complex operations: vector operations, vector operations with memory accesses, and compute or branch instructions with memory operands. The remaining category includes all single operation instructions. We see that on average 27% of the total instructions executed are complex operations.

3.2.3 Calling Convention.

The ISA calling convention describes how subroutines receive parameters from callers and how they return results. Any machine-dependent ISA needs to have its own specifications for passing arguments between subroutines. For example, due to its limited number of registers, x86 pushes all arguments onto the stack before a subroutine is called, resulting in additional stack operations. Other ISAs also require various housekeeping operations for subroutines, and these are also artifacts of the ISA choice, not intrinsic to the behavior of the workload. These additional instructions, mostly stack operations, due

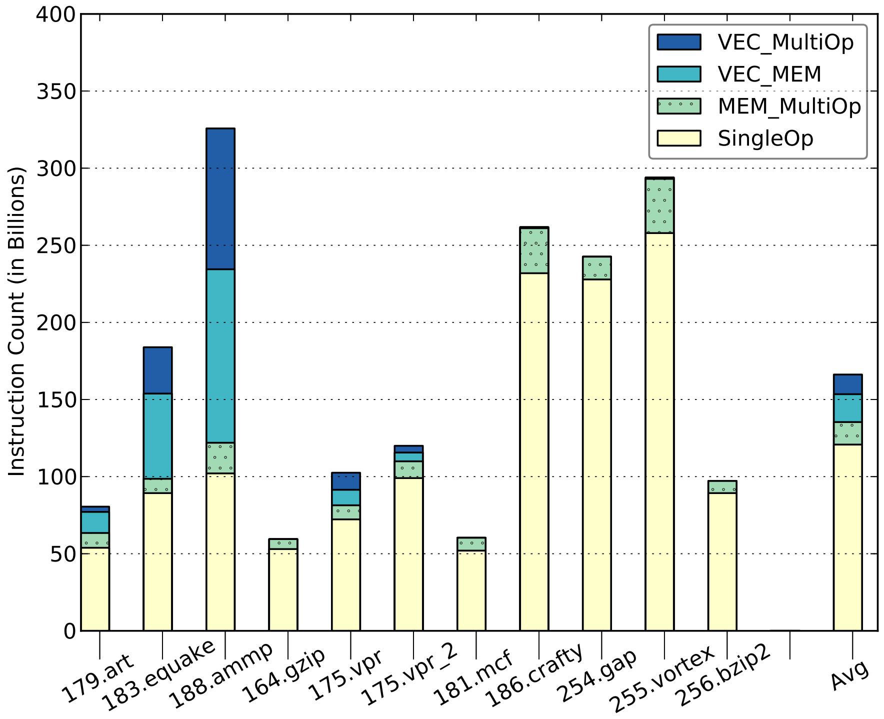


Figure 3.3: Instruction breakdown of complex (top three bars) and single (bottom bar) operation instructions.

to calling conventions could potentially be misleading for specialized architectures that do not have the specific calling convention.

3.3 Methodology and background

To evaluate the importance of performing workload characterization by using machine-independent code representation, we perform both ISA-independent and ISA-dependent analysis.

An ISA-independent representation of code is critical for the development of flexible compiler infrastructures. Fortunately, modern compilers use ISA-independent intermediate representations to bridge high level source languages (e.g., C) to specific ISAs (e.g., Intel x86). Since our requirements for code representation are similar to those of compilers, WIICA leverages the intermediate representation used in compilers to perform its analysis.

WIICA uses the intermediate representation (IR) in both LLVM[12] and ILDJIT[38].[*] Specifically, workload execution is represented by a trace of semantically equivalent IR instructions, instead of ISA-dependent binaries. This dynamic IR instruction trace is generated by executing the IR code with a special-purpose interpreter that emits dynamic IR instructions as it executes them.

3.3.1 Compiler's IR.

ILDJIT is a modular compilation framework that includes both static and dynamic compilers[38]. ILDJIT performs a large set of classical, machine-independent optimizations at the IR level including copy propagation, dead-code elimination, loop-invariant code motion, and the like. When the IR code is fully optimized, it is translated to LLVM's bitcode language and LLVM's back ends are used to optimize the code using machine-dependent optimizations and to generate semantically equivalent machine code.

We customized ILDJIT to implement an ad-hoc interpreter of its intermediate representation that emits IR instructions as they are executed. The IR instructions interpreted are the ones used for translation to the bitcode language. By attaching our interpreter right before the translation to bitcode, we ensure that the IR is fully optimized; however, machine-dependent information is still not used for these optimizations, allowing our analysis to study workload-specific characteristics.

The ILDJIT IR is a linear machine- and ISA-independent representation that includes common operations of high-level programming languages, such as memory allocation (e.g., new, free, newarray) and exception handling (e.g., throw, catch). It is a RISC-like language in which memory accesses are performed through loads and stores. Each instruction has a clear and simple meaning; only scalar variables, memory locations, and the program counter are affected by its execution. The language allows an unbounded number of typed variables (virtual registers), making analysis independent of the number of physical registers. Moreover, parameters of method invocations are always passed by using variables, as in the input source language we use (C), making analysis independent of specific calling conventions. Finally, the data types described in the source language

[*]We use the results obtained with ILDJIT IR for the discussion but the methodology can be extended to LLVM IR as well. The current WIICA distribution uses LLVM IR because the ILDJIT framework is no longer being actively developed.

are preserved in the IR language, making this representation closer to the input language compared to other compiler intermediate representations. Consequently, the three ISA-dependent concerns we are studying - register spilling, complex instructions, and calling conventions - will not appear in the IR being produced.

IR instructions that perform operations among variables require homogeneity among their types: an add operation between variables x and y requires the same type for both x and y (e.g., 32-bit integer). This characteristic leads to instructions that convert values between types. Notice that these conversions are required by the workload as the semantics of operations in the source language specify them. However, some of these conversions are unnecessary if a CISC-like ISA is used instead of the RISC-like IR. Finally, opcodes (e.g., `add`, `mul`) are orthogonal with data types (e.g., integer, floating point). This opcode polymorphism constrains the number of different instructions in the language to 80, allowing an easy parsing of the executed trace.

3.3.2 ISA-Dependent.

To demonstrate the difference between ISA-dependent and -independent analysis, we use the x86 instruction set for the ISA-dependent analysis. The x86 ISA is commonly used in architecture studies, and many program analysis tools are available for workload characterization. For analysis of new x86-based microarchitectures, architects must understand the ISA-specific effects of the architecture since they can have a significant impact on pipeline and memory system design. When considering new heterogeneous architectures with both x86 and specialized cores, it would be natural to use existing workload characterization approaches. However, when performing workload characterization of specialized architectures, our results show that x86 provides a particularly poor starting point. In this study, we compare x86 instruction traces with IR traces. To generate the trace of x86 instructions executed by a workload, we use Pin, a dynamic binary instrumentation tool developed by Intel[108].

3.3.3 Sampling.

Because of storage and processing time constraints, applying some of the analysis presented in this chapter to a full execution trace is impractical. Therefore, we sample execution with SimPoint[148]. We configure SimPoint to generate 10 phases, each of which contains 10 million instructions. Only instructions that belong to the identified phases are emitted and then analyzed.

To make a fair comparison between x86 and IR traces, we sample the execution of the IR trace by configuring SimPoint to use IR instructions. Then we instrument the code to identify the x86 instructions semantically equivalent to the IR code for the identified phases. In this way, we ensure that the same code region is considered for both the IR and x86 analysis.

3.3.4 Benchmark Suite.

We use C benchmarks from SPEC CPU2000 benchmark suite. These benchmarks are translated to CIL bytecode by the compiler GCC4CLI[6] (a branch of GCC), and then they are compiled to IR by ILDJIT. Finally, ILDJIT generates the machine code by relying on LLVM's x86 back end as previously described. ILDJIT currently only supports the 32-bit LLVM back end and all of the results in this section are for 32-bit operations.

3.4 Wordload Characteristics Analysis

3.4.1 Compute

Accelerators often exploit custom functional units that combine multiple operations with predictable control flow in order to execute code more efficiently. An example of this approach is conservation cores[159], which identifies the hot functions in a program's execution and designs hardware accelerators for those functions. To uncover the opportunity to find sequences of operations that are amenable to similar specialization, executed instruction sequences need to be analyzed to detect specific patterns. For such analysis, the way the operations are represented in the instruction trace will have a significant impact on whether certain patterns can be found or not and, subsequently, whether the workload is worth the effort of custom hardware design. In this section, we analyze the instruction breakdown and the most common opcodes found in both x86 and IR. We observe that x86 incurs more overhead for the basic computation performed by the application.

Instruction Breakdown. We start the analysis by categorizing the executed instructions from the IR and x86 code. We split instructions into the following categories:

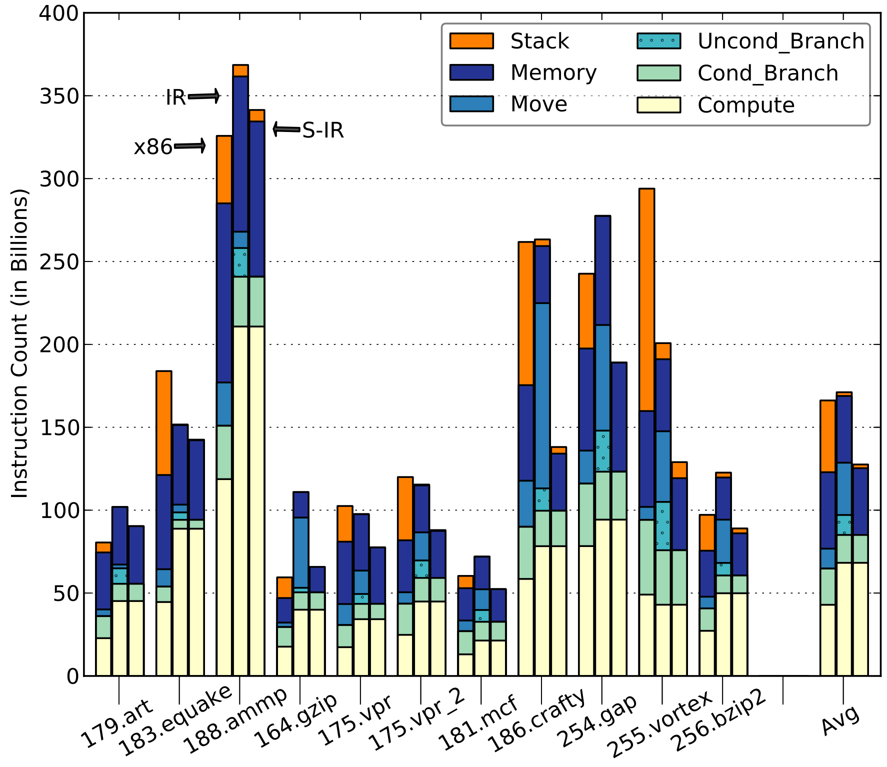


Figure 3.4: The instruction breakdown for x86, IR and Simplified-IR (S-IR).

Stack, Memory, Move (data movement and conversion between registers), Unconditional Branch, Conditional Branch, and Compute. Furthermore, during our implementation, we identified instruction overhead associated with IR characteristics that are not intrinsic to the workloads. One source of such inefficiency is the number of unconditional branch instructions. The ILDJIT compiler does not remove these instructions because the compiler back end performs unconditional branch removal in efficiently. Another source of overhead is data movement and conversion between registers. Such instructions appear in both IR and x86 and are used to support different data types and to simplify optimizations. Figure 3.4 shows this breakdown. For each benchmark, the leftmost bar represents the x86 binary, and the middle bar represents IR. The rightmost bar in Figure 3.4 is what we call Simplified-IR—the IR trace without those two classes of instruction. We also present the normalized instruction breakdown of x86 and simplified IR for the ease of comparison in Figure 3.5. In the following discussion, “IR trace” will refer to

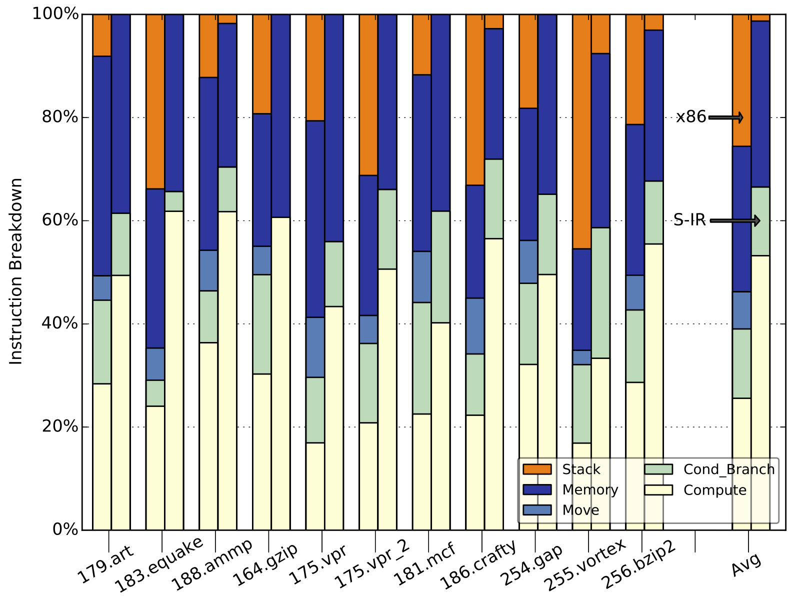


Figure 3.5: The instruction breakdown for x86, IR and Simplified-IR (S-IR).

this simplified IR.

As with the results from LLVM's 32-bit Clang compiler in Figure 3.2, we see that, depending on the application, the number of stack-referencing instructions can be significant. This is represented by the top section of the leftmost bar for every benchmark. For example, almost half of the x86 instructions for 255.vortex use the stack, while the effect is less obvious for benchmarks, such as 179.art. More importantly, the large number of stack accesses results from constraints imposed by the x86 ISA (a small register set) and not from the intrinsic behavior of the program. This is evident from the IR bars for the stack-heavy benchmarks—moving from x86 to the infinite-register IR significantly decreases the number of accesses to the stack. While stack effects can increase the number of executed x86 instructions, CISC x86 instructions can combine multiple primitive operations together. This results in a more compact execution. For example, for benchmarks 164.gzip and 179.art there are more instructions in the IR trace than in the x86 one. The presence of x86-specific effects that both increase and decrease executed instructions makes it even harder to assess ISA-dependent overhead and expose the workload's intrinsic behaviors, further strengthening the case for analysis on the IR level.

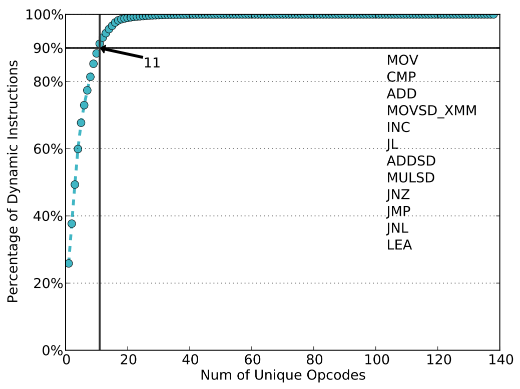


Figure 3.6: x86

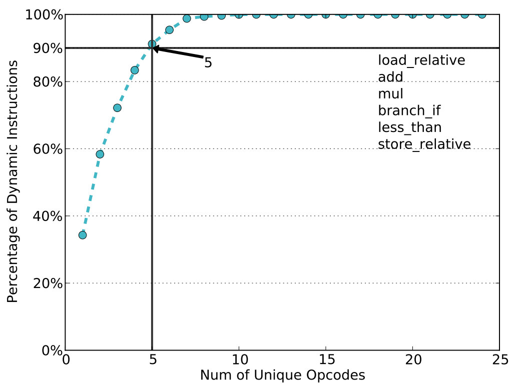


Figure 3.7: IR

Figure 3.8: Cumulative distribution of the number of unique opcodes of 179.art. The intersecting lines show the number of unique opcodes that cover 90% of dynamic instructions.

Opcode Diversity. Our next experiment examines the diversity of the opcodes in the x86 and IR traces. Opcode diversity is relevant since it is related to the complexity of customized functional units in specialized hardware. Fewer and simpler opcodes will simplify the design of such hardware because the functional units will be more modular and reusable. This allows sharing such functional units across various workloads.

To compare x86 and IR analysis, we profile the total number of opcodes and the number of times each single opcode occurs in the program execution. We do not differentiate opcodes based on addressing modes, which reduces the number of required x86 opcodes. Figure 3.8 plots the number of unique opcodes and the percentage of dynamic instructions those opcodes cover for the benchmark 179.art. The dotted line on the plot shows the cumulative distribution of opcodes needed to cover the dynamic execution of the program.

To meaningfully compare x86 and IR, we use a horizontal line to highlight the number of unique opcodes required to cover 90% of the dynamic instructions in Figures 3.6 and 3.7. This metric is meaningful for accelerator studies since it allows comparison of the number of functional unit types needed for different workloads. The horizontal line intersects with the cumulative distribution function to show the required number of opcodes. The x86 results demonstrate that 90% of the execution can be covered by 12 unique opcodes, while the same analysis with IR requires only 6 opcodes (The X axis of Figures 3.6 and 3.7 starts from 0). The right portions of the plots show the top opcodes used for both instruction sets. For x86, two MOV instructions, MOV and MOVSD_XMM, and four different conditional jump instructions are required. Compared with x86, the top opcodes from IR analysis are much clearer—the 6 opcodes are all simple primitives, resulting in a much simpler representation of the actions of the program.

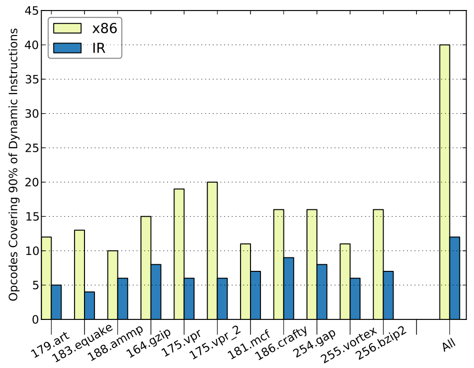


Figure 3.9: Number of unique opcodes to cover 90% of dynamic instructions. "All" represents the global superset.

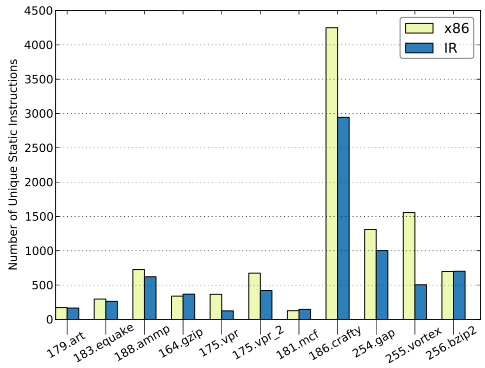


Figure 3.10: Number of unique static instructions to cover 90% of dynamic instructions.

We extend this comparison to all available benchmarks in the suite and show the result in Figure 3.9. Not surprisingly, for each benchmark the x86 trace needs more unique opcodes than the IR trace. Furthermore, the rightmost bar in Figure 3.9 shows the number of unique opcodes required to cover all benchmarks we analyze, computed as a superset of individual benchmark needs. To cover all the benchmarks in x86, 40 unique instruction opcodes are required, but the IR-based analysis uncovers only 12 fundamental primitives. Thus, extracting workload pieces that are amenable to hardware specialization appears significantly easier on the IR level of abstraction.

Static Instructions. The diversity of opcodes represents the different types of fundamental computing blocks that custom hardware might require. Another important metric is the number of static instructions (or the size of the executable code) required to cover the dynamic execution. In a custom design, different sequences of static instructions will lead to more or less complex data flow. As with the metric we use for opcode analysis, we compare the number of unique static instructions required to cover 90% of the dynamic instructions. Figure 3.10 shows that benchmarks that have significant stack overhead (shown in Figure 3.4), such as 186.crafty and 255.vortex, require more unique static instructions, most of which are potentially stack operations. In that case, the x86 characterization hides the truly important instructions, highlighting the stack overhead operations instead. This may skew the identification of hot computation.

3.4.2 Memory

Memory behavior is crucial for workload performance. In the case of hardware specialization, the memory system must be tuned to the workload characteristics in order to realize significant gains in efficiency. In this section, we compare two memory characterization metrics, memory footprint size and memory entropy. We once again discover that ISA-dependent analysis can be significantly misleading and obscure the workloads' intrinsic behavior.

Memory Footprint. The first metric we consider is the size of the data memory that a program uses, including both stack and heap memory. We examine two types of memory footprint. The first one is the full memory footprint—the total size of data memory the program has accessed. It quantifies the overall memory usage. The second metric identifies the "important" memory footprint, which we define as the number of unique memory addresses that cover 90% of dynamic data memory accesses. This metric shows the most frequently used addresses that need to be kept close to the computation.

Figure 3.11 shows the total memory footprint analysis. The Y-axis in this figure is the number of unique memory addresses generated. The x86 and IR memory footprints are nearly the same, because the total working set is intrinsic to the workloads and therefore independent of the program representation.

However, the important memory footprints of x86 and IR, shown in Figure 3.12, are markedly different. In most cases, fewer unique memory addresses are needed for x86 than for IR, despite the fact that their total numbers of unique memory addresses are similar. The reason for this once again lies in frequent accesses to the stack. While the memory space of stack addresses is usually small, these addresses are accessed frequently. When identifying important memory addresses, the few stack addresses that are frequently accessed stand out and dominate the memory behavior. Thus, the important memory addresses found are an artifact of the ISA instead of the program behavior.

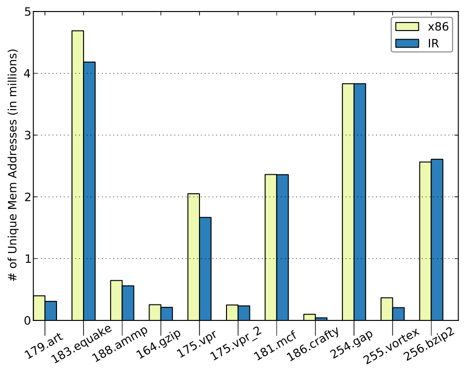


Figure 3.11: Number of unique memory addresses to cover 100% of dynamic memory accesses.

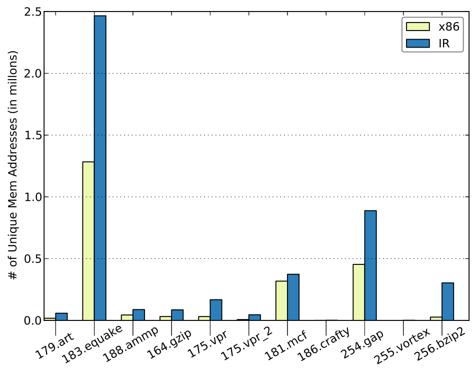


Figure 3.12: Number of unique memory addresses to cover 90% of dynamic memory accesses.

Memory Address Entropy. We introduce *memory address entropy* as a metric that quantifies how easy it is to keep memory data close to the computation. Intuitively, memory address entropy reflects the uncertainty, the lack of predictability, of data addresses accessed by a workload. Thus, it is a metric opposite to memory locality, which is often exploited by custom hardware. Locality measures the regularity of a memory address stream, while entropy measures the lack thereof. We show that ISA-level analysis exposes a lower amount of entropy, leading to false assumptions of memory access regularity.

Entropy. In information theory, entropy[140] is used to measure the randomness of a variable, which is calculated as in Equation 3.1:

$$Entropy = -\sum_{i=1}^{N} p(x_i) * \log_2 p(x_i) \tag{3.1}$$

where $p(x_i)$ is the probability of x_i, and N is the total number of samples of the random variable x. The result, *Entropy*, is a measure of predictability of the next outcome of x. For example, assume the pattern of variable x is regular—always 1. In this case, $p(1) = 1$ and $\log_2 p(1) = \log_2 1 = 0$, so $Entropy = 0$, which means that it is easy to predict x. One the other extreme, if there are N possible outcomes of x occurring equally often,

$p(x_i) = \frac{1}{N}$. According to Equation 3.1,

$$\begin{aligned} Entropy &= -\sum_{i=1}^{N} p(x_i) * \log_2 p(x_i) \\ &= -N * \frac{1}{N} * \log_2(\frac{1}{N}) \\ &= \log_2 N \end{aligned}$$

which is high for large N.

Yen et al. describe the idea of using entropy to represent the randomness of instruction addresses[169]. According to Equation 3.1, in the case of memory entropy, variable x represents the memory addresses that appear in the program execution. The probability $p(x_i)$ is the frequency of a specific memory address x_i. After profiling the unique memory addresses accessed in the workloads and the number of times each address is referenced, we can compute the memory address entropy of the workloads. When the memory entropy is high, the memory access stream is more random and less amenable to architecture techniques that require locality. Conversely, if the entropy is low, memory accesses are regular, easier to predict, and can be effectively addressed by locality techniques.

Global Memory Address Entropy. Global memory entropy describes the randomness of the entire data address stream using all address bits (32 in our case). Figure 3.13 shows the calculated global memory address entropy for both x86 and IR. For each benchmark, the leftmost bar is the global entropy of x86 memory addresses; the rightmost bar is that of IR memory addresses. We can see that the entropy of the x86 address stream is generally lower than that for IR, meaning better temporal locality. To find the reason for the difference, we compute the x86 memory address entropy without the stack addresses, shown in the middle bar. As we can see, after removing the stack addresses, the x86 address entropy is now comparable with the IR memory address entropy. So the higher temporal locality shown in the x86 trace is mostly due to the presence of stack operations. There are two major reasons. First, most of the stack references are from spilled variables. These spilled variables do not exist in IR's memory trace because they are preserved as register operations. Second, it is entirely possible that variables in different phases of the program are mapped to the same stack address. In this case, accesses to these different variables all seem to fetch the same stack address from

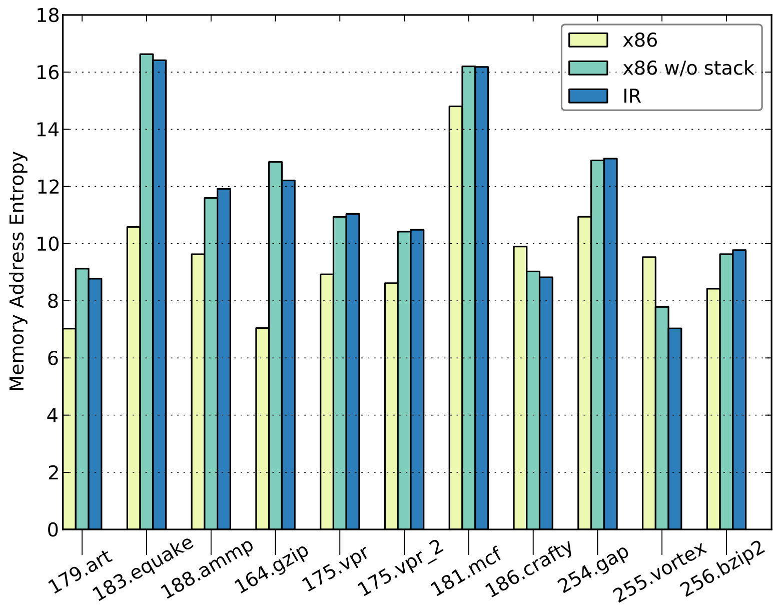


Figure 3.13: Memory address entropy of x86, x86 without stack, and IR traces. Lower values indicate more regularity in the access stream.

the x86 memory trace, leading to higher locality. Such locality is completely an artifact of the ISA, not representing the intrinsic locality of programs.

LOCAL MEMORY ADDRESS ENTROPY. Local memory entropy computes the address entropy using a subset of the address bits. Local entropy can help detect spatial locality in the workloads. For example, we can skip the lower-order bits of the addresses and compute entropy only with the high-order address bits, as seen in Figure 3.14. If the local address entropy with 28 bits, for example, shrinks significantly compared to global entropy, then memory accesses are less random, and significant spatial locality is present. Ignoring the lower order bits reveals spatial locality by grouping those addresses that are close together.

Figure 3.17 shows two examples of the local address entropy when we sweep the number of low order bits ignored from 0 to 10. The two benchmarks, 179.art and 255.vortex, are representative of the patterns we have seen among the rest of the benchmark suite.

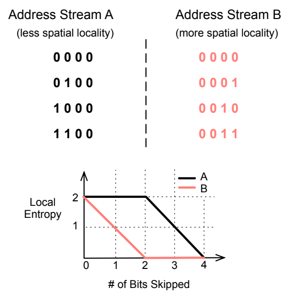


Figure 3.14: Example of Local Entropy

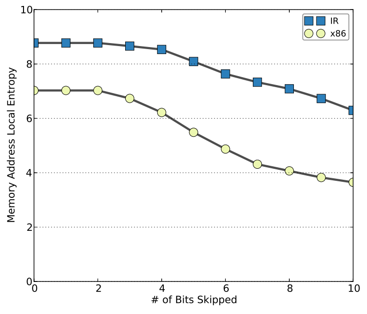


Figure 3.15: 179.art

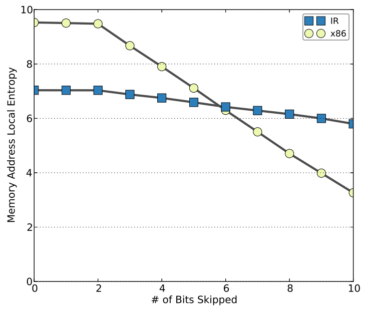


Figure 3.16: 255.vortex

Figure 3.17: Local memory entropy as a function of low-order bits omitted in calculation. A faster dropping curve indicates more spatial locality in the address stream.

For both cases, the local entropy of x86 drops more quickly than for IR. This is obvious for 255.vortex. This is due to the fact that stack addresses are usually in close proximity, which means they usually have good locality. Ignoring the lower-order bits results in steeper drops in entropy for x86. This also shows ISA-dependent analysis will bias the workload characteristics towards better locality due to the impact of stack operations.

3.4.3 Control

Control flow complexity is an important metric for workload characterization. From experience with general purpose processor design, we know that speculative execution is necessary to exploit parallelism. In a heterogeneous architecture, there may be a variety of cores or computing engines with different degrees of support for speculation. To choose the appropriate ones to run the workloads, the control complexity of the workloads needs to be fully understood and not dependent on a specific architecture. In this section, we compare the control complexity analysis of x86 and IR and show that the two analyses are consistent with each other, showing that ISA choice has a minimal effect on a workload's control flow.

Branch Instruction Count. Our first-order control flow analysis counts the number of unique conditional branch instructions that cover 90% of the dynamic branches. This is similar to the unique opcode analysis but focuses on branch instructions. This is important for hardware specialization because it measures the number of control flow

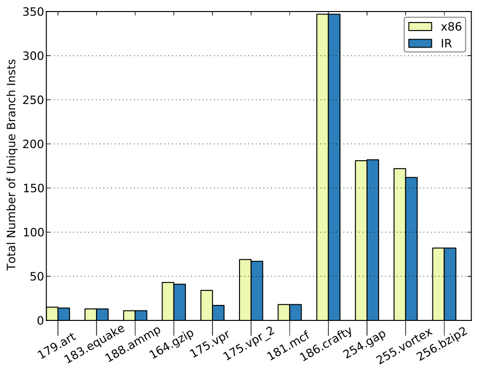


Figure 3.18: Number of unique branch instructions to cover 90% of dynamic branches.

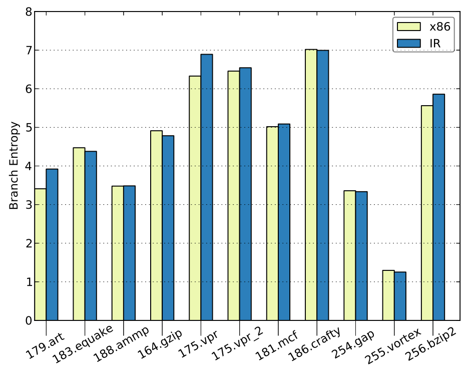


Figure 3.19: Branch entropy per workload. Lower values imply better branch predictability.

decisions that must be handled in a design.

As Figure 3.18 shows, the number of unique branch instructions required to cover 90% dynamic branches is consistent between x86 and IR. The two sets of bars track each quite closely. This implies that ISA choice does not have a significant impact on the number of branch instructions generated, which mostly depends on the way programs are written.

Branch History Entropy. Another important metric is control flow predictability, which is intrinsic to the workload. Generally speaking, if the branch taken patterns are more regular and less random, branches are easier to predict. In this sense, the regularity of the branch behavior will indicate the predictability of the control flow. Based on this intuition, Yokota proposed the idea of *branch history entropy* using Shannon's information entropy idea to represent a program's predictability[170].

We use a string of bits to encode taken or not taken branch outcomes. In this sense, the program, as the producer of the sequence, can be viewed as an information source and we can compute the entropy of the information source to represent the regularity of branch behavior. In our implementation, we use a sequence of n consecutive branch results as the random variable and compute the entropy of the benchmarks. The results are shown in Figure 3.19. We can see that the branch entropy from x86 and IR are also quite similar. This shows that both ISA-dependent analysis and ISA-independent analysis fully expose the program's control behavior. This matches our intuition that ISA does not affect control flow significantly.

3.5 Putting it all together

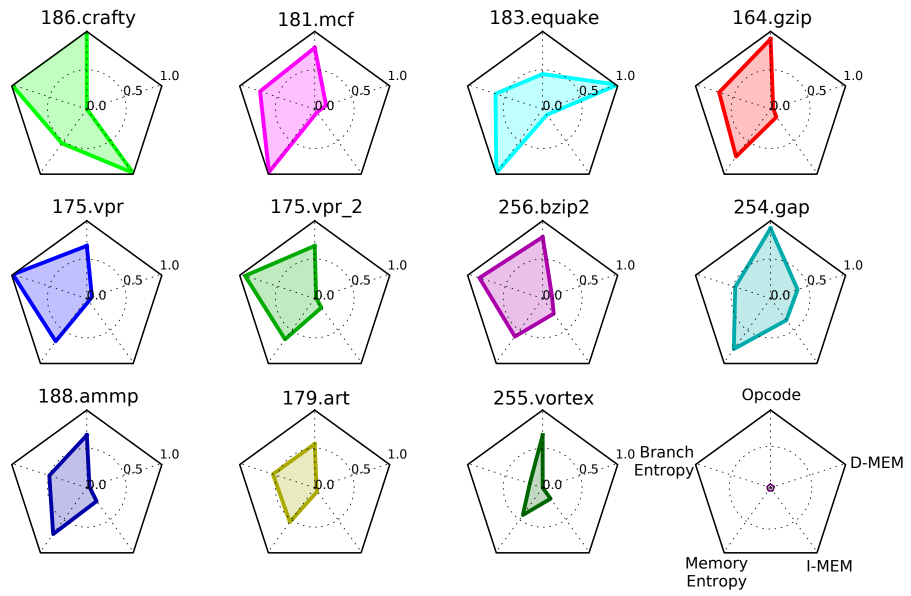


Figure 3.20: Comparison of five ISA-independent metrics across SPEC benchmarks, ordered by the area of the polygon. The lower right kiviat plot provides the legend, and smaller values indicate more regularity in the metric.

We compare the eleven SPEC benchmarks with five ISA-independent metrics from our analysis: the number of opcodes, the value of branch entropy, the value of memory entropy, the unique number of static instructions (I-MEM), and the unique number of data addresses (D-MEM). In terms of specialized architecture design, smaller values for each of these metrics indicate more regularity in the benchmarks and better opportunity to exploit specialization. For each metric, we choose the maximum value across all the benchmarks and for each benchmark we plot the relative value with respect to this maximum value. Figure 3.20 shows kiviat plots for all benchmarks, in which each axis represents one of the ISA-independent characteristics. The plot in the lower right corner of the figure provides a legend for the individual axes. The kiviat plots are ordered by the area of the resulting polygon. With an equal weighting of the five characteristics, area provides a rough approximation for overall benchmark regularity (smaller area is more regular). We observe different behavior across the benchmark suite. For example, 255.vortex demonstrates regularity across all the metrics, while 186.crafty has relatively low regularity in most of the dimensions. These insights will be helpful for specialized architecture designers to identify the opportunities for acceleration.

Although ISA-independent workload characterization is useful for designers to understand intrinsic workload characteristics, it does not quantitatively evaluate the cost and benefits of building a specific accelerator for a given kernel. This has motivated us to think how to rapidly evaluate different design choices of accelerator designs without going through the time consuming design flow. Building upon the WIICA project, next chapter introduces Aladdin, a power-performance-area simulator for fixed-function accelerators, enabling large design space exploration for specialized architectures.

"Love truth, but pardon error."

Voltaire, 1738.

4 Aladdin: Pre-RTL, Power-Performance-Area Accelerator Modeling

Current research in accelerator analysis relies on RTL-based synthesis flows to produce accurate timing, power, and area estimates. Such techniques not only require significant effort and expertise but are also slow and tedious to use, making large design space exploration infeasible. To overcome this problem, this chapter discusses Aladdin, a pre-RTL, power-performance accelerator modeling framework and demonstrates its application to system-on-chip (SoC) simulation. Aladdin estimates performance, power, and area of accelerators within 0.9%, 4.9%, and 6.6% with respect to RTL implementations. Integrated with architecture-level core and memory hierarchy simulators, Aladdin provides researchers an approach to model the power and performance of accelerators in an SoC environment.

4.1 Introduction

As we near the end of Dennard scaling, traditional performance and power scaling benefits based on technology improvements no longer exist. At the same time, transistor density improvements continue; the result is the dark silicon problem in which chips now have more transistors than a system can fully power at any point in time[63,159]. To overcome these challenges, hardware acceleration, in the form of datapath and control circuitry customized to particular algorithms or applications, has surfaced as a promising approach, as it delivers orders of magnitude performance and energy benefits compared to general purpose solutions. Customized architectures composed of CPUs, GPUs, and accelerators are already in widespread use in mobile systems and are beginning to emerge in servers and desktops.

The natural evolution of this trend will lead to a growing volume and diversity of customized accelerators in future systems, where a comprehensive assessment of potential benefits and trade-offs across the entire system will be critical for system designers. However, current customized architectures contain only a handful of accelerators, as large design space exploration is currently infeasible due to the lack of a fast simulation infrastructure for accelerator-centric systems.

Computer architects have long been developing and leveraging high-level power[35,102,141] and performance[26,32] simulation frameworks for general-purpose cores and GPUs[29,101]. In contrast, current accelerator-related research primarily relies on creating RTL implementations, a tedious and time-consuming process. It takes hours, if not days, to generate, simulate, and synthesize RTL to get the power and performance of a single accelerator design, even with the help of high-level synthesis (HLS) tools. Such a low-level, RTL infrastructure cannot support architecture-level design space exploration that sweeps parameters across traditional general-purpose cores, accelerators, and shared resources such as cache hierarchies and on-chip networks. Hence, there is a clear need for a high-level design flow that abstracts RTL implementations of accelerators to enable broad design space exploration of next-generation customized architectures.

We introduce Aladdin, a pre-RTL, power-performance simulator designed to enable rapid design space search of accelerator-centric systems. This framework takes high-level language descriptions of algorithms as inputs and uses dynamic data dependence graphs (DDDG) as a representation of an accelerator without generating RTL. Starting with an unconstrained program DDDG, which corresponds to an initial representation of accelerator hardware, Aladdin applies optimizations as well as constraints to the graph to create a realistic model of accelerator activity. We rigorously validated Aladdin against RTL implementations of accelerators from both handwritten Verilog and a commercial HLS tool for a range of applications, including accelerators in Memcached[103], HARP[164], NPU[65], and a commonly used throughput-oriented benchmark suite, SHOC[58]. Our results show that Aladdin can model performance within 0.9%, power within 4.9%, and area within 6.6% compared to accelerator designs generated by traditional RTL flows. In addition, Aladdin provides these estimates over 100× faster.

Aladdin captures accelerator design trade-offs, enabling new architectural research directions in heterogeneous systems composed of accelerators, general-purpose cores, and the shared memory hierarchy seen in today's mobile SoCs and for future customized architectures; we demonstrate this capability by integrating Aladdin with a full cache hierarchy model and DRAMSim2[134]. Such infrastructure allows users to explore customized and shared memory hierarchies for accelerators in a heterogeneous environment. In a case study with the GEMM benchmark, Aladdin uncovers significant, high-level, design trade-offs by evaluating a broader design space of the entire system. Such analysis results in more than 3× performance improvements compared to the conventional approach of designing accelerators in isolation.

4.2 Background and Motivation

Hardware acceleration exists in many forms, such as analog accelerators[30,155], static[49,65,78,103,129,159,164] and dynamic datapath accelerators[50,74,76], and programmable accelerators, such as GPUs and DSPs. In this work, we focus on static datapath accelerators. Here we discuss the design flow, design space, and state-of-the-art research infrastructure of datapath accelerators, all in order to illustrate the challenges associated with current accelerator research and why a tool like Aladdin opens up new research opportunities for architects.

4.2.1 Accelerator Design Flow

The current accelerator design flow requires multiple CAD tools, which is inherently tedious and time-consuming. It starts with a high-level description of an algorithm, then designers either manually implement the algorithm in RTL or use HLS tools, such as Xilinx's Vivado HLS[20], to compile the high-level implementation (e.g., C/C++) to RTL. It

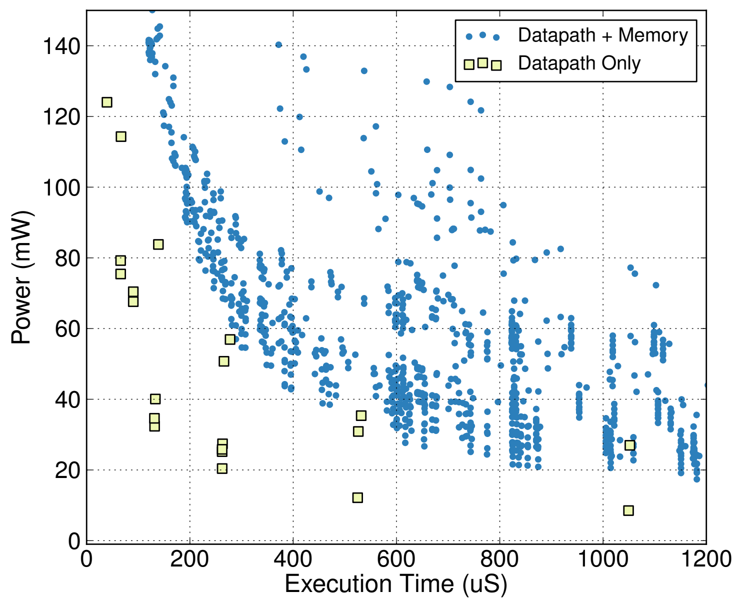


Figure 4.1: GEMM design space w/ and w/o memory hierarchy.

takes significant effort to write RTL manually, the quality of which highly depends on designers' expertise. Although HLS tools offer opportunities to automatically generate the RTL implementation, extensively tuning C-code is still necessary to meet design requirements. After generating RTL, designers must use commercial CAD tools, such as Synopsys's Design Compiler and Mentor Graphics' ModelSim, to estimate power and cycle counts.

In contrast, Aladdin takes unmodified, high-level language descriptions of algorithms, generating a DDDG representation of accelerators, which accurately models the cycle-level power, performance, and area of realistic accelerator designs. As a pre-RTL simulator, Aladdin is orders of magnitude faster than existing CAD flows.

4.2.2 Accelerator Design Space

Despite the application-specific nature of accelerators, the accelerator design space is large given a range of architecture- and circuit-level alternatives. Figure 4.1 illustrates a large power-performance design space of accelerator design points for the GEMM workload from the SHOC benchmark suite. The square points were generated from a commercial HLS flow sweeping datapath parameters, including loop-iteration parallelism,

	Novel Accelerator Design	Accelerator Datapath Trade-offs	Heterogeneous SoC Trade-offs
handwritten RTL	Buffer-int-Cache[66], Memcached[98,103], Sonic Millip3De[136], HARP[164]	Inadequate	Inadequate
HLS	LINQits[48], Convolution Engine[129], Conservation Cores[159]	Cong[55], Liu[104], Reagen[132]	Inadequate

Table 4.1: Accelerator Research Infrastructure

pipelining, array partitioning, and clock frequency. However, HLS flows generally provision a fixed latency for all memory accesses, implicitly assuming local scratchpad memory fed by DMA controllers.

Such simple designs are not well suited for capturing data locality or interactions with complex memory hierarchies. The circle points in Figure 4.1 were generated by Aladdin integrated with a full cache hierarchy model and DRAMSim2, sweeping not only datapath parameters but also memory parameters. By doing so, Aladdin exposes a rich design space that incorporates the realistic memory penalties in terms of time and power, impractical with existing HLS tools alone. Section 4.5 further demonstrates the importance of accelerator datapath and memory co-design using Aladdin.

4.2.3 State-of-the-art Accelerator Research Infrastructure

The ITRS predicts hundreds to thousands of customized accelerators by 2022[18]. However, state-of-the-art accelerator research projects still contain only a handful of accelerators because of the cumbersome design flow that inhibits computer architects from evaluating large accelerator-centric systems. Table 4.1 categorizes accelerator-related research projects in the computer architecture community over the past 5 years based on the means of implementation (handwritten RTL vs. HLS tools) and the scope of possible design exploration.

We see that researchers have been able to propose novel implementations of accelerators for a wide range of applications, either writing RTL directly or using HLS tools

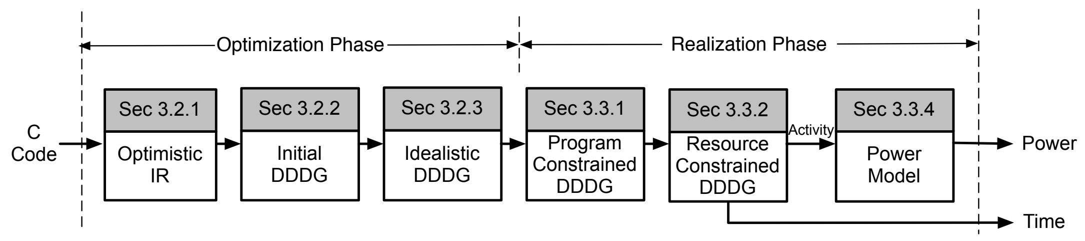


Figure 4.2: The Aladdin Framework Overview.

despite the time-consuming process. With the help of the HLS flow, we have begun to see studies evaluating design trade-offs in accelerator datapaths, which are otherwise impractical using handwritten RTL. However, as discussed in Section 4.2.2, HLS tools cannot easily navigate large design spaces of customized architectures. This inadequacy in infrastructure has confined the exploratory scope of accelerator research.

4.2.4 Contributions

In summary, this work makes the following contributions:

1. We present Aladdin, a pre-RTL, power-performance simulator for fixed-function accelerators using dynamic data dependence graphs (Section 4.3).

2. We perform rigorous validation of Aladdin against handwritten RTL implementations and a commercial HLS design flow. We show that Aladdin can model the behavior of recently published accelerators[103,164,65] and typical accelerator kernels[58] (Section 4.4).

3. We demonstrate a large design space exploration of customized architectures, enabled by Aladdin, identifying high-level accelerator design trade-offs (Section 4.5).

4.3 The Aladdin Framework

4.3.1 Modeling Methodology

The foundation of the Aladdin infrastructure is the use of dynamic data dependence graphs (DDDG) to represent accelerators. A DDDG is a directed, acyclic graph, where nodes represent computation and edges represent dynamic data dependences between nodes. The dataflow nature of hardware accelerators makes the DDDG a good candidate to model their behavior. Figure 4.2 illustrates the overall structure of Aladdin, starting from a C description of an algorithm and passing through an *optimization* phase, described in Section 4.3.2, where the DDDG is constructed and optimized to derive an idealized representation of the algorithm. The idealized DDDG then passes to a *realization* phase, discussed in Section 4.3.3, that restricts the DDDG by applying realistic program dependences and resource constraints. User-defined configurations allow wide design space exploration of accelerator implementations. The outcome of these two phases is a pre-RTL, power-performance model for accelerators.

Aladdin uses a DDDG to represent program behavior so that it can take arbitrary C code descriptions of an algorithm—without any modifications—to expose algorithmic parallelism. This fundamental feature allows users to rapidly investigate different algorithms and accelerator implementations. Due to its optimistic nature, dynamic analysis has been previously deployed in parallelism research exploring the limits of ILP [27,69,130,162] and recent modeling frameworks for multicore processors[72,87]. These studies sought to quickly measure the upper bound of performance achievable on an ideal parallel machine[95]. Our work has two main distinctions from these efforts. First, previous efforts model traditional Von Neumann machines where instructions are fetched, decoded, and executed on a fixed, but programmable architecture. In contrast, Aladdin models a vast palette of different accelerator implementation alternatives for the DDDG; the optimization phase incorporates typical hardware optimizations, such as removing memory operations via customized storage inside the datapath and reducing the bitwidth of functional units. The second distinction is that Aladdin provides a realistic power-performance model of accelerators across a range of design alternatives during its realization phase, unlike previous work that offered an upper-bound performance estimate.

In contrast to dynamic approaches, parallelizing compilers and HLS tools use program dependence graphs (PDG)[52,70] that statically capture both control and data dependences[68,75]. Static analysis is inherently conservative in its dependence analysis, because it is used for generating code and hardware that works in all circumstances and is built without run-time information. A classic example of this conservatism is the enforcement of false dependences that restrict algorithmic parallelism. For instance, programmers often use pointers to navigate arrays, and disambiguating these memory references is a challenge for HLS tools. Such situations frequently lead to designs that are more sequential compared to what a human RTL programmer would develop. Therefore, although HLS tools offer the opportunity to automatically generate RTL, designers still need to extensively tune their C code to expose parallelism explicitly (Section 4.4). Thus, *Aladdin is different from HLS tools*; Aladdin is simply a realistic, accurate representation of accelerators, whereas HLS is burdened with generating actual, correct hardware.

This section describes details of the optimization phase (Section 4.3.2) and realization phase (Section 4.3.3) of Aladdin. We then discuss how to integrate Aladdin with memory systems (Section 4.3.4) and limitations of the approach (Section 4.3.5).

4.3.2 Optimization Phase

The optimization phase forms an idealized DDDG that only represents the fundamental dependences of the algorithm. An idealized DDDG for accelerators must satisfy three requirements: (a) express only necessary computation and memory accesses, (b) only capture true read-after-write dependences, and (c) remove unnecessary dependences in the context of customized accelerators. This section describes how Aladdin's optimization phase addresses these requirements.

Optimistic IR

Aladdin builds the DDDG from a dynamic instruction trace, where the choice of the ISA significantly impacts the complexity and granularity of the nodes in the graph. In fact, a trace using a machine-specific ISA contains instructions that are not part of the program but produced due to the artifacts of the ISA[142], *e.g.*, register spills. To avoid such artifacts, Aladdin uses a high-level, machine-independent intermediate representation (IR) provided by the ILDJIT compiler[38]. ILDJIT IR is optimistic because it allows an unlimited number of registers, eliminating additional instructions generated due to stack overheads and register spilling. The IR contains 80 opcodes ranging from simple primitives, *e.g.*, add and multiply, to complex operators, *e.g.*, sine and square root, so that we can easily detect the functional units needed based on the program's IR trace and model them using pre-characterized hardware. We use a customized interpreter for the ILDJIT IR to emit fully-optimized IR instructions in a trace file. The trace includes dynamic instruction information such as opcodes, register IDs, parameter data types, and parameter data values. We also profile the dynamic addresses of memory operations.

Initial DDDG

Aladdin analyzes both register and memory dependences based on the IR trace. Only true read-after-write data dependences are respected in the initial DDDG construction. This DDDG is optimistic enough for the purpose of ILP limit studies but is missing several characteristics of hardware accelerators; the next section discusses how Aladdin idealizes the DDDG further.

Idealized DDDG

Hardware accelerators have considerable flexibility to customize datapaths for application-specific features, which is not modeled in the initial DDDG. Such customization can change the attributes of the datapath, as in the case of bitwidth reduction where functional units can be tuned to the value range of the problem. Aladdin also removes operations that are not required for hardware implementations. For example, to reduce memory bandwidth, small, frequently accessed arrays, such as filters, can be stored directly in registers inside the datapath instead of in external memory. Cost models are used to automatically perform all of these transformations.

We categorize our optimizations into node-level, loop-level, and memory-level transformations to produce an idealized DDDG representation.

Node-Level Optimization. In addition to bitwidth analysis, we also model other node-level optimizations, such as strength reduction and tree-height reduction, by changing the nodes' attributes and performing standard graph transformations[56].

Loop-Level Optimization. The initial DDDG captures the true dependences between successive iterations of the loop index variables, which means each index variable can only be incremented once per cycle. Such dependence constraints do not apply to hardware accelerators or parallel processors since it is entirely possible that they can initiate multiple iterations of a loop simultaneously[156]. Aladdin removes all dependences between loop index variables, including basic and derived induction variables, to expose loop parallelism.

Memory Optimization. The goal is to remove unnecessary load/store operations. In addition to the memory-to-register conversion example described above, Aladdin also performs store-load forwarding inside the DDDG, which eliminates load operations by buffering data in internal registers within hardware accelerators. This is different from store-load forwarding in general-purpose CPUs, where the load operation must still be executed[138].

Extensibility Hardware design is open-ended, and Aladdin can be extended to incorporate other accelerator-specific optimizations, analogous to adding new microarchitectural structures to CPU simulators. We demonstrate this extensibility by considering CAM hardware to optimize data matching. A CAM is an example of a custom circuit structure that is often used to accelerate hash tables in network routers and datatype specific accelerators[173]. Unlike software, CAMs can automatically compare a key against all of the entries in one cycle. On the other hand, large CAMs are power hungry, resulting in an energy trade-off when hash tables reach a certain size. Aladdin incorporates CAMs into its customization strategy by automatically replacing software-managed hash tables with CAM. Aladdin can detect a linear search for a key by looking for chained sequential memory look-ups and comparison. Section 4.4.2 demonstrates an example with a Memcached accelerator in which CAMs are used as a victim cache to a regular hash table during hash conflicts[103].

4.3.3 Realization Phase

The realization phase uses program and resource parameters, defined by users, to constrain the idealized DDDG generated in the optimization phase.

Program-Constrained DDDG

The idealized DDDG optimistically assumes that hardware designers can eliminate all control and false data dependences at design time. Aladdin's realization phase models actual control and memory dependences to create the program-constrained DDDG.

Control Dependence. The idealized DDDG does not include control dependences, assuming that branch outcomes can be known in advance and operations can start before branches are resolved, which is unrealistic even for hardware accelerators. The costs and benefits of control flow speculation for accelerators have not been extensively studied yet, and one solution to minimize control dependences relies on predicated execu-

Parameters	Example Range
Loop Rolling Factor	[1::2::Trip count]
Clock Period (ns)	[1::2::6]
FU latency	Single-Cycle, Pipelined
Memory Ports	[1::2::64]

Table 4.2: Realization Phase User-Defined Parameters, i::j::k denotes a set of values from i to k by a stepping factor j.

tion to simultaneously execute both taken and not taken paths until branch resolution[97]. While this approach minimizes serialization, the cost of speculation is high—it requires hardware resources that grow exponentially with the number of outstanding branches. Aladdin models control dependence by bringing code from the not-taken path into the program-constrained DDDG to account for additional power and resources. Aladdin is flexible enough to model the costs of different mechanisms for handling control flow. For energy efficiency, Aladdin models one outstanding branch at a time, serializing control dependences for multiple simultaneous branches.

Memory Dependence. The idealized DDDG optimistically removes all false memory dependences between dynamic instructions, keeping true read-after-write dependences. This is realistic for memory accesses with addresses that can be resolved at design time. However, some algorithms have input-dependent memory accesses, *e.g.*, histogram, where different inputs result in different dynamic dependences. Without runtime memory disambiguation support, designers have to make conservative assumptions about memory dependences to ensure correctness. To model realistic memory dependences, the realization phase includes memory ambiguation that constrains the input-dependent memory accesses by adding dependences between all dynamic instances of a load-store pair, as long as a true memory dependence is observed for any pair. This is similar to the dynamic dependence profiling approach adopted by parallelization efforts[72,91].

Resource-Constrained DDDG

Finally, Aladdin accounts for user-specified hardware resource constraints, a subset of which are shown in Table 4.2. Users specify the type and size of hardware resources in an input configuration file. Aladdin then binds the program-constrained DDDG onto the hardware resources, leading to the resource-constrained DDDG. Aladdin can easily sweep resource parameters to explore the design space of an algorithm, which is fast because only resource constraints need to be applied for each design point. These resource parameters are set with respect to the following three factors: loop rolling, loop pipelining, and memory ports.

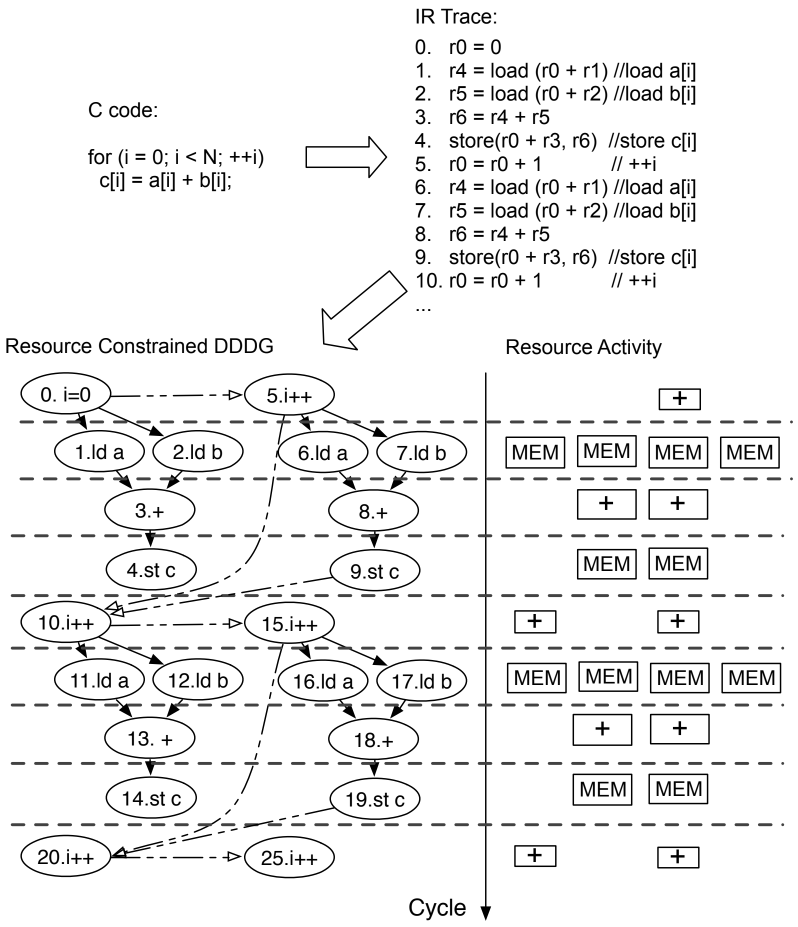
Loop Rolling. The optimization phase removes dependences between loop index variables, assuming completely unrolled loops that execute all iterations in parallel. In reality, for loops with large trip counts, this leads to large resource requirements. Aladdin's loop rolling factor re-rolls loops by adding dependences between loop index variables.

Loop Pipelining. The DDDG representation fully pipelines loop iterations by default, though sometimes pipelined implementation leads to high resource requirements as well as high power consumption. Aladdin offers users the option to turn off loop pipelining by adding dependences between the entry and exit nodes of successive loop iterations.

Memory ports. The number of memory ports constrains the data transfer rate between the accelerator datapath and the closest memory hierarchy, generally either a scratchpad memory or L1 cache. Aladdin uses this parameter to abstractly model the number of memory requests the datapath can issue concurrently, and Section 4.3.4 discusses how the memory ports interface with memory simulators.

An Example

Figure 4.3 illustrates different phases of Aladdin transformations using a microbenchmark as an example. After the IR trace of the C code has been produced, the optimization and realization phases generate the resource-constrained DDDG that models accelerator behavior. In this example, we assume the user wants an accelerator with a factor-of-2 loop-iteration parallelism and without loop pipelining. The solid arrows in the DDDG are true data dependences, and the dashed arrows represent resource constraints, such as loop rolling and turning off loop pipelining. The horizontal dashed lines represent clock cycle boundaries. The corresponding resource activities are shown to the right of the DDDG example. We see that the DDDG reflects the dataflow nature of the accelerator. Aladdin can accurately capture dynamic behavior of accelerators without having to



C code:

```
for (i = 0; i < N; ++i)
  c[i] = a[i] + b[i];
```

IR Trace:

```
0.  r0 = 0
1.  r4 = load (r0 + r1) //load a[i]
2.  r5 = load (r0 + r2) //load b[i]
3.  r6 = r4 + r5
4.  store(r0 + r3, r6)  //store c[i]
5.  r0 = r0 + 1          // ++i
6.  r4 = load (r0 + r1) //load a[i]
7.  r5 = load (r0 + r2) //load b[i]
8.  r6 = r4 + r5
9.  store(r0 + r3, r6)  //store c[i]
10. r0 = r0 + 1          // ++i
...
```

Figure 4.3: C, IR, Resource Constrained DDDG, and Activity.

generate RTL by carefully modeling the opportunities and constraints of the customized datapath in the DDDG.

Power and Area Models

We now describe the construction and application of Aladdin's power and area models to capture the resource requirements of accelerators.

Power Model. To accurately model the power of accelerators, we need: (a) precise activities and (b) accurate power characterization of different DDDG components. We uniquely characterize switching, internal, and leakage power from Design Compiler for each type of DDDG node (multipliers, adders, shifters, etc.) and registers. The characterization accounts for different timing requirements, bitwidths, and switching activity. Switching and internal power are due to capacitive charging/discharging of output load and internal transistors of the logic gates, respectively. While switching and internal power are both dynamic, we found internal power weakly dependent on activity because internal nodes can switch without the gate output switching.

We construct a detailed power model by synthesizing microbenchmarks that exercise the functional units. Our microbenchmarks cover all of the compute instructions in IR so that there is a one-to-one mapping between nodes in the DDDG and functional units in the power model. We synthesize these microbenchmarks using Synopsys's Design Compiler in conjunction with a commercial 40nm standard cell library to characterize the switching, internal, and leakage power of each functional unit. This characterization is fully automated in order to migrate easily to new technologies.

Aladdin's power modeling library also accounts for cell selection variances during netlist synthesis. Different pipeline stages within a datapath contain varying amounts of logic and, in order to meet timing requirements, different standard cells and logic implementations of functional units are often selected at synthesis time. Aladdin approximates the impact of cell selection by training the model for a variety of timing constraints, using a first-order model to choose the correct design. This also accounts for logic flattening that Design Compiler performs across small collections of functional units.

Area Model. To accurately model area, we construct an area library similar to the previously described power library for each DDDG component. This model was obtained using the same set of microbenchmarks to characterize the area for each functional unit as well as for registers.

Cycle-Level Activity. Figure 4.4 shows the cycle-level resource activity for one implementation of the FFT benchmark. Aladdin accurately captures the distinct phases of FFT. The number of functional units required is estimated using the maximum number of parallel functional units per cycle for each program phase; this approximation provides the power and area models with the total resources allocated to the accelerators. The cycle-level activity is an input to the power model to represent the dynamic activity of the accelerators.

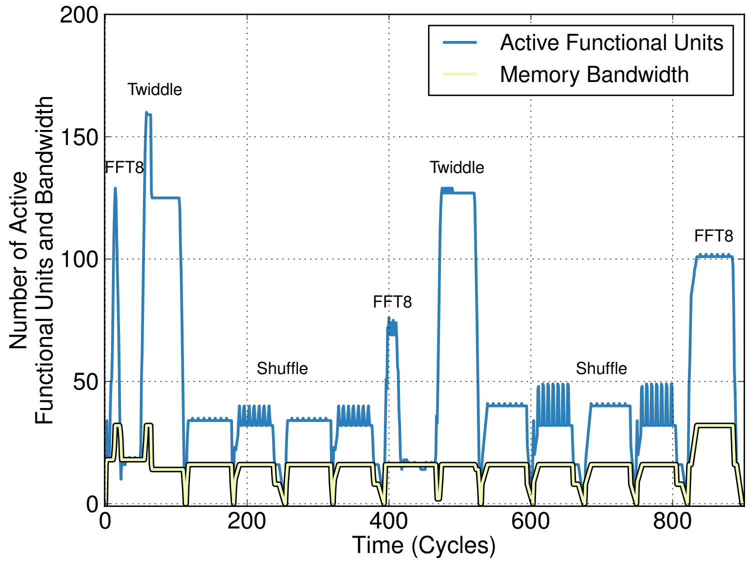


Figure 4.4: Cycle-by-Cycle FU and Memory Activity of FFT.

4.3.4 Integration with Memory System

Aladdin can easily integrate with architectural cache and memory simulators to model their behavior with a particular memory hierarchy. Within the context of memory hierarchy for accelerators, we discuss three types of memory models with which Aladdin can integrate.

Ideal Memory guarantees that all memory requests can be serviced in one cycle, which is realistic only for a system with small memory size. Aladdin models the ideal memory system by assuming load and store nodes in the DDDG take one cycle.

Scratchpad Memory is commonly used in accelerator-centric systems where accelerator designers explicitly manage memory accesses so that each request has a fixed latency. However, this approach requires a detailed understanding of workload memory characteristics. This potentially increases design time but leads to more efficient implementation. Aladdin can take a parametrized memory latency as an input to model the latency of load and store operations matching the characteristics of scratchpad memory.

Cache Hierarchy applies a hardware-managed cache system to capture locality of the accelerated workload. Such a cache hierarchy relies on the hardware to exploit the locality of the workload, potentially easing the design of systems with a large number of accelerators. On the other hand, a cache introduces variable memory latency. Existing cache simulators can be integrated with Aladdin to evaluate how variable latency memory accesses affect accelerator behaviors.

In order to integrate with a cache hierarchy, the accelerator must include certain mechanisms to react to possible cache misses. Aladdin models several approaches to handle this variable latency, which resemble pipeline control mechanisms in general-purpose processors. The simplest policy is local or global pipeline stalls on miss events. We also consider a more complex mechanism for non-blocking behavior in which a new loop iteration is started when a miss occurs, and only the loop ID is stored for re-execution when the miss resolves.

Memory Power Model. The memory power model is based on a commercial register file and SRAM memory compiler that accompanies our standard cell library. We have compared the memory power model to CACTI[163] and found consistent trends, but we retain the memory compiler model for consistency with the standard cell library.

4.3.5 Limitations

Algorithm choices. Aladdin does not automatically sweep different algorithms. Rather, it provides a framework to quickly explore various hardware designs of a particular algorithm. This means designers can use Aladdin to quickly and quantitatively compare the design spaces of multiple algorithms for the same application to find the most suitable algorithm choice.

Input Dependent. Like other dynamic analysis frameworks[84,106], Aladdin only models operations that appear in the dynamic trace, which means it does not instantiate hardware for code not executed with a specific input. For Aladdin to completely model the hardware cost of a program, users must provide inputs that exercise all paths of the code.

Input C code. Aladdin can create a DDDG for any C code. However, in terms of modeling accelerators, C constructs that require resources outside the accelerator, such as system calls and dynamic memory allocation, are not modeled. In fact, understanding how to handle such events is a research direction that Aladdin facilitates.

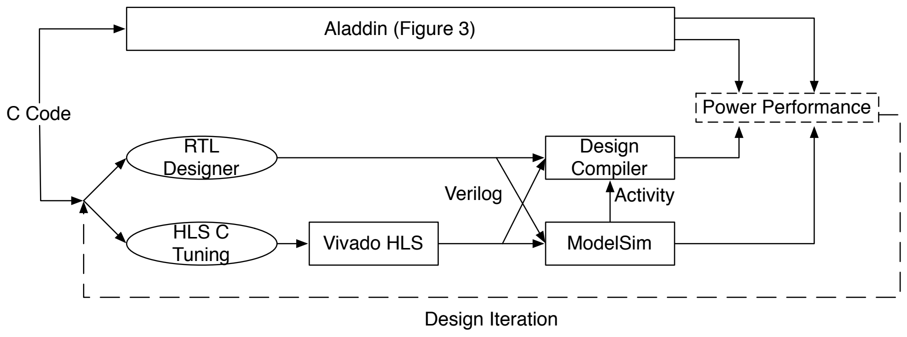


Figure 4.5: Validation Flow.

4.4 Aladdin Validation

We begin this section with a detailed description of the traditional RTL design flow and workloads used to validate Aladdin. Validation results show Aladdin has modest error rates within 0.9% for performance, 4.9% for power, and 6.5% for area. Aladdin generates the design space more than 100× faster than the traditional RTL-based flow.

4.4.1 Validation Flow

Figure 4.5 outlines the methodology used to validate Aladdin. The power and area estimates of Aladdin are compared against synthesized Verilog generated by Design Compiler using commercial 40nm standard cells. Aladdin's performance model is validated against ModelSim Verilog simulations. The SAIF activity file generated from ModelSim is fed to Design Compiler to capture the switching activity at the gate level. To generate Verilog, we either hand-code RTL or use Xilinx's Vivado HLS tool. The RTL design flow is an iterative process and requires extensive tuning of both RTL and C code.

HLS Tuning

We use HLS to generate the accelerator design space for SHOC benchmarks to demonstrate Aladdin's ability to explore a large design space of an accelerator's datapath, which is infeasible with handwritten RTL. To produce high-quality Verilog, HLS requires significant tuning of the input C code to expose parallelism and remove false dependences. In contrast, Aladdin produces the power-performance optimal design points without modifying the input C code.

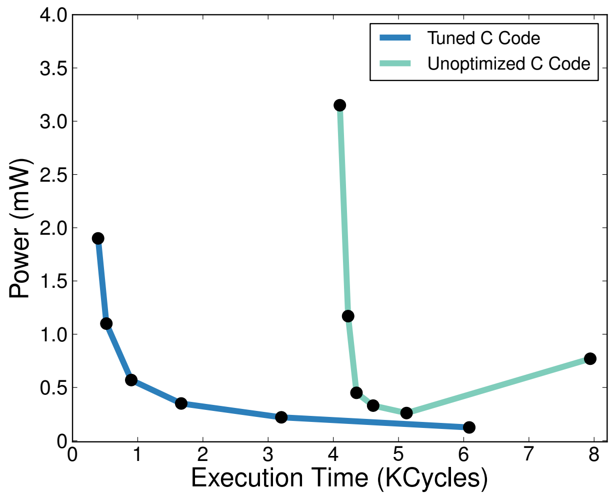


Figure 4.6: Unoptimized vs. Tuned Scan.

Figure 4.6 demonstrates the quantitative difference that code quality can have on power and performance by comparing Pareto frontiers of optimized and unoptimized versions for the *Scan* benchmark. Both curves were generated by sweeping loop unrolling factors, memory bandwidth, and resource sharing and applying loop pipelining, similar to the parameters discussed in Section 4.3. The unoptimized C code hits a performance wall at around 4000 cycles where neither increasing bandwidth nor loop parallelism yields better performance but continues to burn more power. The reason is that when striding over a partitioned array being read from and written to in the same cycle, though accessing different elements of the array, the HLS compiler conservatively adds loop-carried dependences. This in turn increases the iteration interval of loop pipelining, limiting performance. To overcome HLS's conservative assumptions, we partition the array differently, which consequently simplifies the access patterns to resolve false dependences. Similar tuning was necessary to generate well-performing designs for each of the SHOC benchmarks, which are then used to validate Aladdin in Section 4.4.3.

4.4.2 Applications

We implemented a collection of benchmarks, both by hand and using HLS, to validate Aladdin. HLS enabled the validation of the Pareto optimal designs for the SHOC benchmarks, overcoming the impracticality of hand coding each design point. We also validate

Aladdin against handwritten RTL for benchmarks ill-suited for HLS. Examples are taken from recently published accelerator research: NPU[65], Memcached[103], and HARP[164].

SHOC

The SHOC benchmark suite is representative of many typical accelerator workloads, which include compute intensive benchmarks where functional units often dominate execution time and power, *e.g.*, Stencil, as well as memory-bound workloads, *e.g.*, Sort, stressing Aladdin's modeling capabilities across multiple dimensions. To ensure valid, well performing HLS results, we carefully tuned each implementation as described earlier. By sweeping loop unrolling factors and resource constraints such as memory bandwidth, a large design space of accelerator datapaths for each benchmark is generated.

Single Accelerators

In some instances, the expressiveness of C limits the ability for HLS to reasonably match hand-coded RTL. Therefore, we hand coded RTL for HARP, NPU, and Memcached to further demonstrate Aladdin's modeling capabilities. For Aladdin, we rely on generic C implementations that describe the behavior of each accelerator.

HARP is a partitioning accelerator for big data[164]. Essentially, given a stream of inputs, it uses a pipeline of comparators to check each input against a splitter value at each stage and categorize the inputs. HARP is a control-intensive workload where its activity highly depends on the input values, which makes it a good candidate to exercise Aladdin's ability to model control behavior. Our handwritten Verilog for HARP properly expresses the pipelined comparisons. Aladdin was able to match the Verilog implementation through the loop rolling and pipelining parameters.

NPU is a network of individual neurons connected through a shared bus, which communicate with each other in a carefully orchestrated, compiler-generated pattern. The design hinges on an input FIFO to buffer computations. Although HLS has FIFO support, the ability to finely share data efficiently between compute engines is a shortcoming of most HLS tools. An individual neuron was implemented in Verilog, and a synthetic input was used to stimulate the neuron. Aladdin's memory-to-register transformation successfully captures such FIFO-type structure.

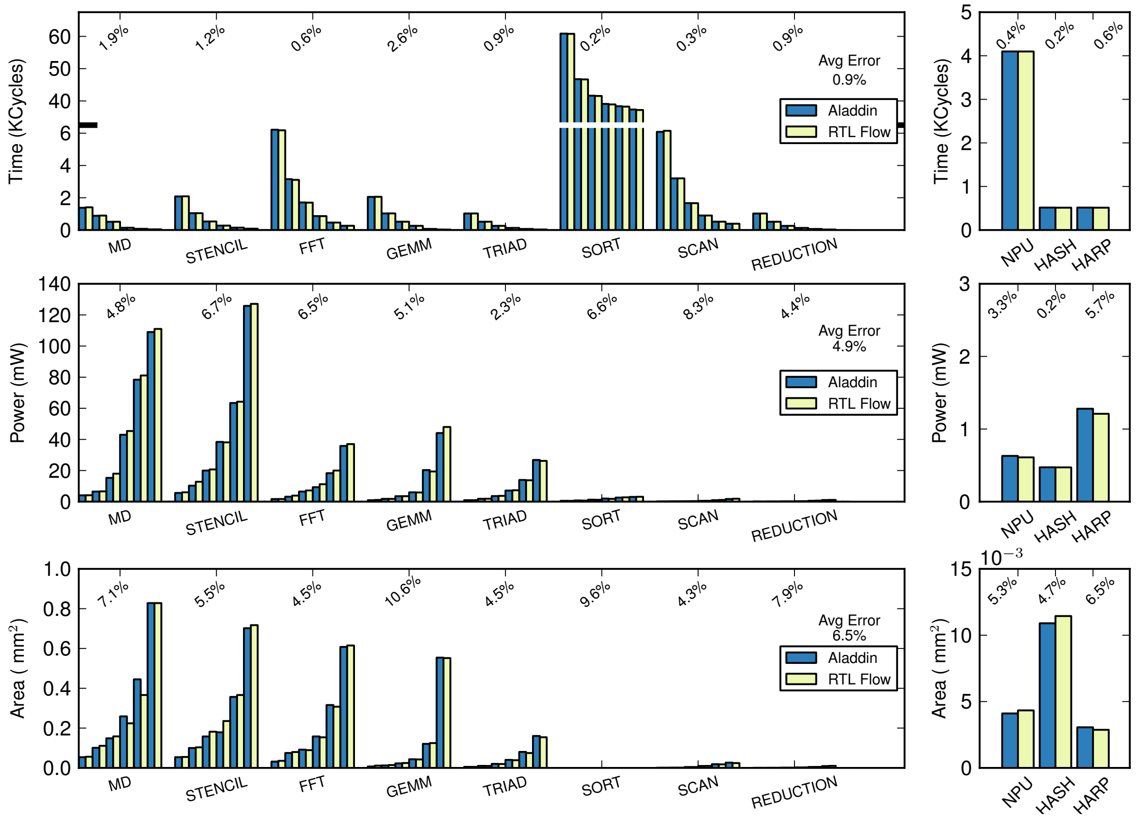
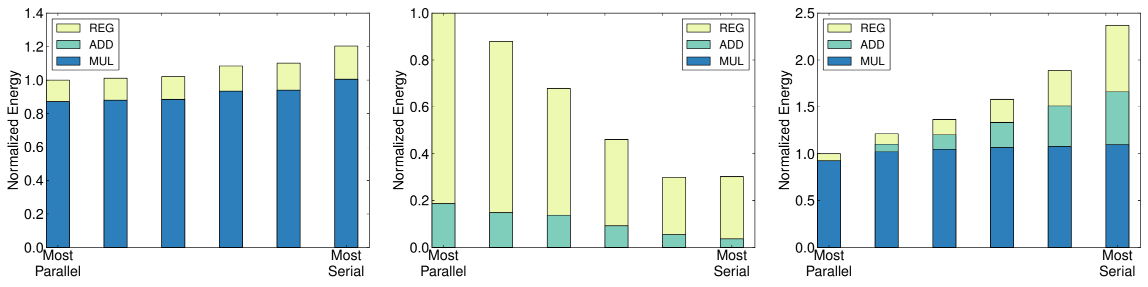


Figure 4.7: Performance (top), Power (middle), and Area (bottom) Validation.

Memcached is a distributed key-value store system, whose functionality is mapped to a hash function and CAM lookup. Given an input key, a hash accelerator computes the value using a hash algorithm[103]. The value is then used to index four SRAMs whose content is compared to the input key to determine a hit. If one of the SRAMs returns a match, it returns that SRAM's data. On a miss, the value is sent to a CAM where all possible locations of the key are checked in parallel and the correct value is returned. This benchmark serves two purposes—to demonstrate Aladdin's ability to model variable bitwidth computations (the hash function) and to model a different customization strategy (CAM).



(a) Triad **(b)** Sort **(c)** Stencil

Figure 4.8: Energy Characterization of SHOC.

4.4.3 Validation

Figure 4.7 shows that Aladdin accurately models performance, power, and area compared against RTL implementations across all of the presented benchmarks with average errors of 0.9%, 4.9%, and 6.5%, respectively. For each SHOC workload, we validated six points on the Pareto frontier, *e.g.*, points in Figure 4.6. The SHOC validation results show Aladdin accurately models entire design spaces, while for single accelerator designs, Aladdin is not subject to HLS shortcomings and can accurately model different customization strategies.

Pareto Analysis The Pareto optimal designs of the SHOC benchmarks reveal interesting program characteristics in the context of hardware accelerators. Bars in Figure 4.8 correspond to six designs along each benchmark's Pareto frontier, which were also used for validation. In each graph, the leftmost bar is the most parallel, highest performing design while the rightmost bar is the most serial and lowest performing design. For each design, we calculate energy using power and performance estimates from Aladdin. Aladdin's detailed power model enables energy breakdowns for adders, multipliers, and registers. The six bars of each benchmark are normalized to the leftmost bar to facilitate comparisons.

Each of the three benchmarks in Figure 4.8 exhibits different energy trends across the Pareto frontier. Triad, shown in Figure 4.8a, demonstrates good energy proportionality, meaning more parallel hardware leads to better performance with a proportional power cost. In contrast, Sort has a strong sequential component such that energy increases for more parallel designs without improving performance. Finally, while the multiplier en-

	Hand-Coded RTL	HLS	Aladdin
Programming Effort	High	Medium	N/A
RTL Generation	Designer Dependent	37 mins	
RTL Simulation Time	5 mins		
RTL Synthesis Time	45 mins		
Time to Solution per Design	87 mins		1 min
Time to Solution (36 Designs)	52 hours		7 mins

Table 4.3: Algorithm-to-Solution Time per Design.

ergy for Stencil shows similar energy proportionality to Triad, the adders and registers required for loop control are amortized with more parallelism. Non-intuitively, this leads to better energy efficiency for these faster designs.

4.4.4 Algorithm-to-Solution Time

Aladdin enables rapid design space exploration of accelerator designs. Table 4.3 quantifies the differences in algorithm-to-solution time to explore a design space of the FFT benchmark with 36 points. Compared to traditional RTL flows, Aladdin skips the time-consuming RTL generation, synthesis, and simulation process. On average, it takes 87 mins to generate a single design using the RTL flow but only 1 min for Aladdin, including both of Aladdin's optimization phase (50 seconds) and realization phase (12 seconds). However, because Aladdin needs to perform the optimization phase only once for each algorithm, this optimization time can be amortized across large design spaces. Consequently, it only takes 7 mins to enumerate the full design space with Aladdin compared to 52 hours with the RTL flow. The HLS RTL generation time per design is comparable to that reported by other researchers[104].

4.5 Case Study: GEMM Design Space

We now present a case study that demonstrates how Aladdin enables architecture research and why it is invaluable to future heterogeneous SoC designs. We focus our analysis on GEMM as it has complex memory behavior and analyze a problem size of 196 KB. In this case study, we present:

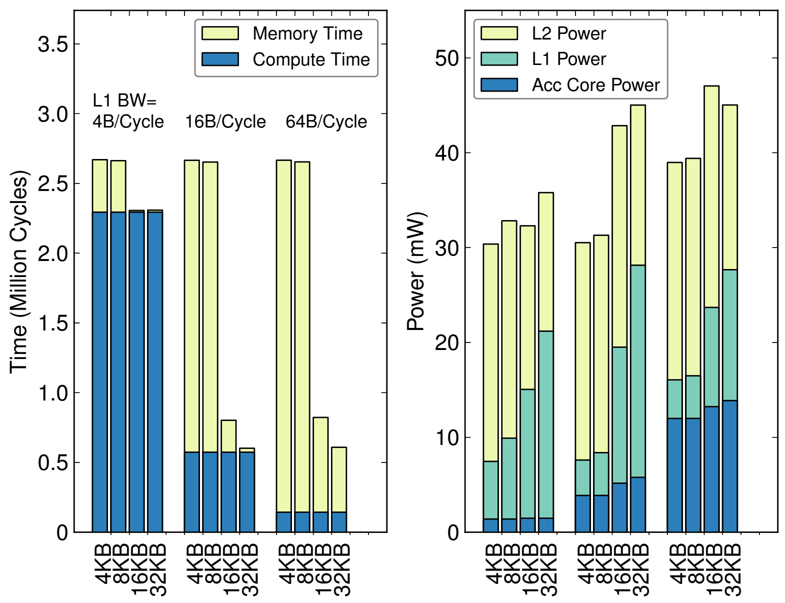


Figure 4.9: GEMM Time and Power Decomposition

1. Execution Time Decomposition: Understand design trade-offs of an accelerator's execution time with respect to *compute time* and *memory time*.
2. Accelerator Design Space: Characterize the design space of GEMM accelerators, including memory hierarchy, to understand how different parameters affect the design space.
3. Heterogeneous SoC: Demonstrate the impact of resource contention in an SoC-like system of a single accelerator, resulting in different optimal designs that would be unknown without system-level analysis.

4.5.1 Execution Time Decomposition

So far, Aladdin has been evaluated as a standalone accelerator simulator with an ideal memory hierarchy (one cycle memory access latency). However, it is not always possible to retrieve data in one cycle in real designs with large problem sizes. The efficiency of accelerators highly depends on the memory system. To quantify the impact of a memory system on accelerators, we integrate Aladdin with a standard cache simulator and the DRAMSim2 memory simulator[134].

We divide the accelerator's execution time into *compute time* and *memory time*. Compute time is defined as the execution time of an accelerator when there is only one cycle memory latency. Memory time is defined as cycles lost to a non-ideal memory, which includes both memory bandwidth and memory latency constraints.

In order to decompose the accelerator's execution time, we run simulations with both an ideal memory and a realistic memory hierarchy including L1, L2, and DRAM. The compute time is the execution time with ideal memory; the delta of execution times between the two simulations is the memory time to get data into accelerators[37].

Type	Parameters	Values
Core	Blocking Factor	[16, 32]
L1	L1 Bandwidth (Bytes/Cycle)	[4::2::128]
	L1 Size (KB)	[4::2::32]
	MSHR Entries	[4::2::64]
L2	L2 Bandwidth (Bytes/Cycle)	[4::2::128]
	L2 Size (KB)	[64::2::256]
	L2 Assoc	16

Table 4.4: Single Accelerator Design Space, where i::j::k denotes a set of values from i to k by a stepping factor of j.

Table 4.4 lists all of the parameters in the design space. In this section, we focus on the bandwidth and size of L1. Figure 4.9 shows the execution time and power breakdown of the GEMM benchmark when sweeping L1 size and bandwidth. On the left of Figure 4.9, we observe that memory time takes a significant portion of the execution time, especially as L1 bandwidth increases. With the same L1 bandwidth, execution time decreases as the L1 size increases from 8 KB to 16 KB; this phenomenon occurs because 8 KB is not large enough to hold the blocked data size (a 32×32 matrix).

The plot on the right shows the power breakdown of the accelerator datapath, L1, and L2. The accelerator datapath power increases with L1 bandwidth, because higher bandwidth enables more parallel implementations. As L1 size increases, its power also increases as accesses become more expensive. At the same time, L2 power decreases because more accesses are coalesced by the L1, lowering the L2 cache's activity. In fact, cache power consumes more than half of the total power, even for more parallel designs

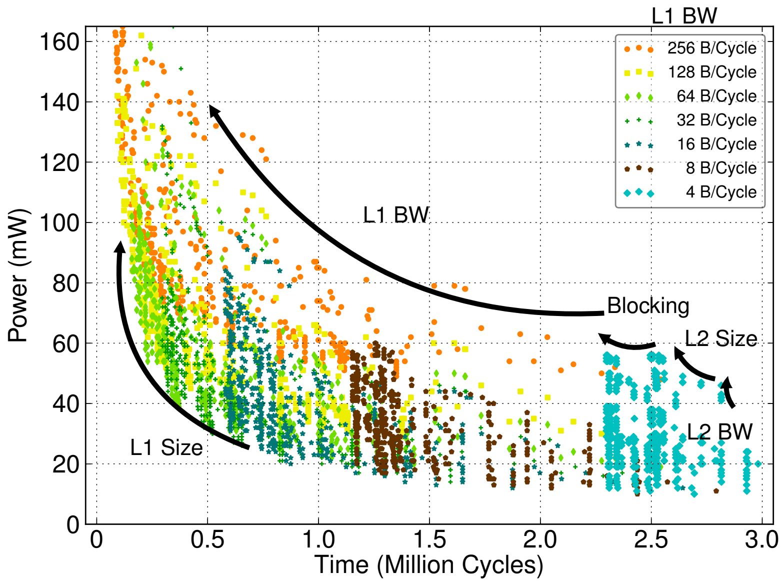


Figure 4.10: GEMM design space.

where datapath power is significant. Therefore, design efforts focusing on the accelerator datapath alone do not alleviate memory power, which dominates the overall power cost.

4.5.2 Accelerator Design Space

Section 4.5.1 explored a subset of the design space for accelerators and memory systems. Here, we use Aladdin to explore the comprehensive design space with parameters in Table 4.4. Figure 4.10 plots the power and execution time of the GEMM accelerator designs resulting from the exhaustive sweep. The design space contains several overlapping clusters of similar designs. The arrows in Figure 4.10 identify correlations in power/performance trends with respect to each parameter. For example, GEMM experiences substantial performance benefits from a larger L1 cache, but with a significant power penalty. In contrast, increasing L2 size only modestly increases both power and performance.

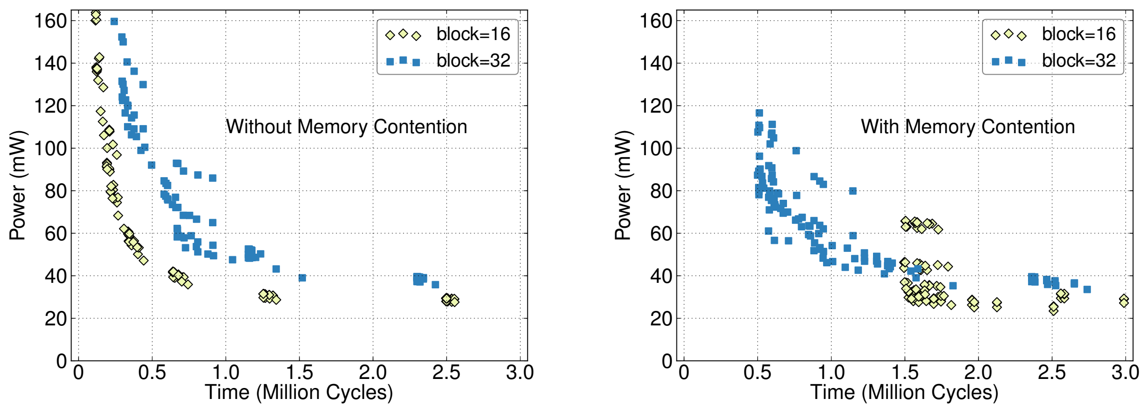


Figure 4.11: Design Space of GEMM without and with contention in L2 cache.

4.5.3 Resource-Sharing Effects in Heterogeneous SoC

In a heterogeneous system, shared resources, such as a last-level cache, can be accessed by both general-purpose cores and accelerators. We consider the case of a heterogeneous system consisting of a shared 256 KB L2 cache, one general-purpose core, and a GEMM accelerator with a private 16 KB L1 cache. From an accelerator designer's perspective, an important algorithmic parameter is the blocking factor of GEMM; a larger blocking factor exposes more algorithmic parallelism, however, achieving good locality requires a larger cache.

Figure 4.11(left) shows the accelerator design space without memory contention from the general purpose core. We modulate the algorithmic blocking factor and find that a blocking factor of 16 is always better than 32 with respect to both power and performance. This occurs because a 16 KB L1 cache is large enough to capture the locality of the blocking factor 16 but not 32. Therefore, it is preferable to build the accelerator with blocking factor 16 when there is no contention for shared resources.

To model resource contention between the general-purpose core and the accelerator, we use Pin[107] to profile an x86 memory trace and then use the trace to issue requests that pollute the memory hierarchy while simultaneously running the accelerator. The design space for the accelerator under contention is shown in Figure 4.11(right). We see that performance degrades for both blocking factors of 16 and 32 due to pollution in the L2 cache; however, blocking factor 32 suffers much less than blocking factor 16. When there is contention, capacity misses increase for the shared L2 cache, which incurs large main memory latency penalties. With a larger blocking factor, the accelerator requires fewer references to the matrices in total and, thus, fewer data requests from the L2 cache. Consequently, the effects of resource contention suggest building an accelerator with a larger blocking factor, where the accelerator performance can achieve around 0.5 million cycles. On the other hand, without considering the contention, designers may pick a design with blocking factor 16, the highest performance of which is 1.5 million cycles in the contention scenario. Such design choice leads to a 3× performance degradation. Aladdin can easily evaluate these types of system-wide accelerator design trade-offs, a task that is not tractable with other current accelerator design tools.

"You cannot see the forest for the trees."

5 gem5-Aladdin: Accelerator-System Co-Design

This chapter extends the work of the previous chapter and highlights that co-designing of the accelerator microarchitecture with the system in which it belongs is critical to balanced, efficient accelerator microarchitectures. We find that data movement and coherence management for accelerators are significant, yet often unaccounted, components of total accelerator runtime, resulting in misleading performance predictions and inefficient accelerator designs. To explore the design space of accelerator-system co-design, this chapter introduces gem5-Aladdin, an SoC simulator that captures dynamic interactions between accelerators and the SoC platform, and validates it to within 6.5% against real hardware. The co-design studies show that the optimal energy-delay-product (EDP) of an accelerator microarchitecture can improve by up to 7.4× when system-level effects are considered compared to optimizing accelerators in isolation.

5.1 Introduction

Accelerators are often designed as standalone intellectual property (IP) blocks that communicate with the rest of the system using a Direct Memory Access (DMA) interface. This modularity simplifies IP design and integration with the rest of the system, leaving tasks such as data movement and coherency management to software device drivers. As a result, the costs of these overheads are hard to predict and accommodate at accelerator design time. Our detailed characterization of accelerator behavior shows that the combination of just these two effects can occupy over 40% of the total runtime. When it comes to accelerator design, architects must take a holistic view of how they interact in the overall system, rather than designing them in isolation.

Fundamentally, all systems should be designed in a way that balances the bandwidth of the memory interface with the amount of compute throughput. An overly aggressive design will have more computational units than the memory interface can supply, leading to wasted hardware and additional leakage power. We show that three major system-level considerations that strongly affect balanced accelerator design are the local memory system interface, cache coherency management, and behavior under shared resource contention.

The typical local memory interface is DMA, a push-based system that requires software to set up bulk transfers and manage coherency. An alternative is to embed a hardware-managed cache with the accelerator design, leading to a fine-grained pull-based memory system that loads data on-demand and transparently handles coherency state. Despite these conveniences, caches are rarely used in accelerators due to hardware overheads leading to power and area penalties. However, there has been growing interest from industry in providing coherent accelerator cache interfaces[36,119,152]. We investigate the system-level considerations for both approaches to understand when each is preferable.

Such studies require a detailed simulation infrastructure for heterogeneous accelerator-rich platforms such SoCs. There is a wide selection of CPU simulators[32,39] and standalone accelerator simulators such as Aladdin[145]. However, existing SoC simulators are unable to model dynamic interactions between accelerators and the memory system[53]. In this paper, we describe gem5-Aladdin, which integrates the gem5 system simulator with the Aladdin accelerator simulator to enable simulation of SoCs with complex accelerator-system interactions. We validate gem5-Aladdin against the Xilinx Zynq platform and achieve less than 6.5% error.

We demonstrate that co-designing accelerators with system-level considerations has two major ramifications for accelerator microarchitectures that are not yet fully understood in the literature. First, datapaths should be less aggressively parallel, producing more balanced designs and improved energy efficiency compared to accelerators designed in isolation. Second, the choice of local memory interfaces is highly dependent on the dy-

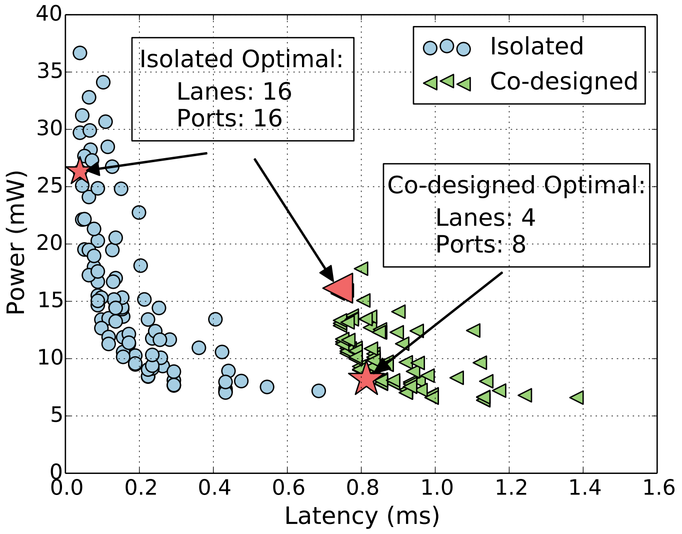


Figure 5.1: Isolated and co-designed EDP optimal design points for stencil3d.

namic memory characteristics of the accelerated workload, the system architecture, and the desired power/performance targets. We show that accelerator-system co-design can improve energy-delay-product by up to 7.4× and on average by 2.2×.

5.2 Motivation and Background

We use the term "accelerator" to refer to an application-specific hardware block. Accelerators are comprised of multiple customized datapath lanes and customized local memories. Each lane is a chain of functional units controlled by finite state machines. When the local memory is comprised of scratchpads, each scratchpad can be partitioned into smaller arrays to increase memory bandwidth to the lanes. Such accelerators are representative of recent academic proposals[49,65,78,103,129,159,164] and commercial designs[19,33,94].

5.2.1 Co-design: A Motivating Example

To demonstrate the differences between isolated vs. co-designed accelerators, we perform a design sweep exploration for both scenarios on a 3D stencil kernel. We sweep compute parallelism and scratchpad partitioning. Compute parallelism is described by the number of datapath lanes. Figure 5.1 shows these two design spaces.

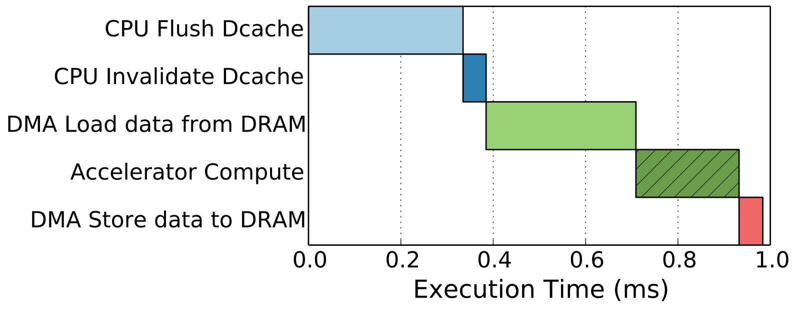
We consider an accelerator designed in isolation to be one that focuses design optimization on the computation phase. This design space (blue circles) leans towards more parallel, power-hungry designs, as exemplified by the isolated energy-delay-product (EDP) optimal design point. But if we account for effects such as initial data movement, the design space (green triangles) shifts dramatically towards the lower right, preferring less parallel designs at lower power. If we take the isolated EDP optimal design and then apply these system effects, we find that it is quite different from the co-designed EDP optimal point. Unaccounted data movement becomes a significant part of total runtime, making aggressively parallel datapaths unnecessary.

5.2.2 Typical CPU-Accelerator Communication

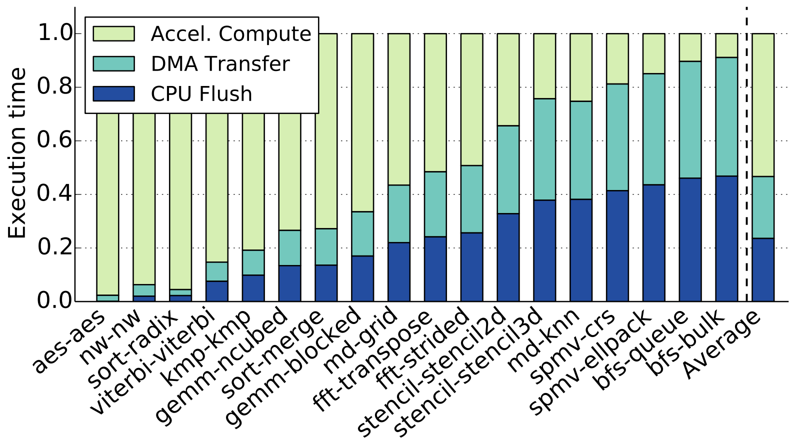
The existence of the difference between the two design spaces is due to how CPUs and accelerators traditionally communicate data. In this typical flow, DMA is the transfer mechanism, but typical DMA implementations can only access main memory or LLC, so the CPU first flushes all input data from private caches and invalidates the region used to store return data[168]. Then it programs a DMA transaction into the DMA engine and initiates the transfer. The accelerator begins execution after receiving all the data and streams its output data via DMA back to main memory when it is done. The CPU, having invalidated that memory region from its caches, can now access the return data correctly.

For many benchmarks, this flow works well. DMA is efficient at copying large blocks of data, and accelerators whose compute-to-memory ratios are large are well served by DMA. However, for other workloads with more irregular memory access patterns, this flow can impose severe overheads, because the accelerator must wait to receive all the data before it can begin computation. As an example, Figure 5.2a shows the execution timeline for a 16-lane implementation of an `md-knn` accelerator (a k-nearest-neighbor molecular dynamics), running on a Xilinx Zynq platform. As shown, the accelerator's computation consumes only about 25% of the total cycles, with the rest of the time spent preparing and moving data. We expand this study in simulation for all the MachSuite benchmarks and find that about half of them are compute-bound and the other half are data-movement-bound, as shown in Figure 5.2b.

Clearly, DMA is not an optimal solution for some workloads. One alternative, as mentioned earlier, is to replace push-based DMA with pull-based hardware-managed caches.



(a) `md-knn` execution time on the Zynq platform



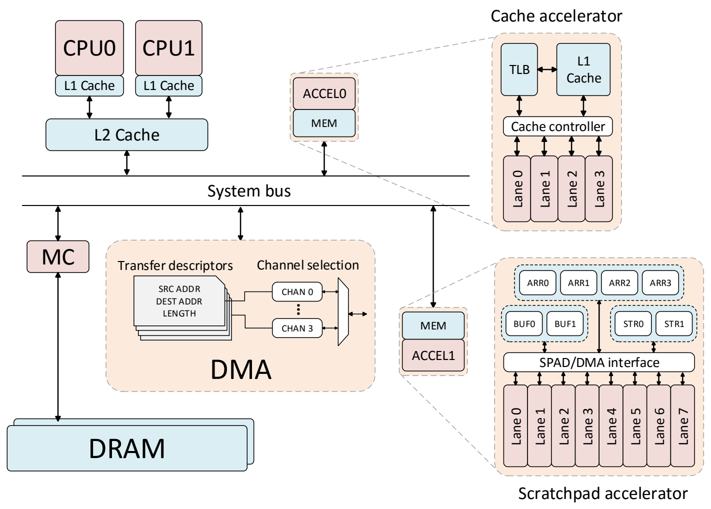
(b) Breakdown of flush, DMA, and compute time in MachSuite for 16-way parallel designs.

Figure 5.2: Data movement overheads on MachSuite.

In recent years, the scope of workloads that we desire to accelerate has widened from dense computational kernels to more irregular applications which could benefit from a less rigid memory system. Although caches have seldom been used for accelerators, the increased workload diversity motivates a more comprehensive study of new CPU-accelerator communication strategies.

5.3 Modeling infrastructure

Figure 5.3 shows an example of an SoC, including general-purpose cores, memory controllers, a DMA engine, and different types of fixed-function accelerators, all of which are



Design Parameter	Values
Datapath lanes	1, 2, 4, 8, 16
Scratchpad partitioning	1, 2, 4, 8, 16
Data transfer mechanism	DMA/cache
Pipelined DMA	Enable/disable
DMA-triggered compute	Enable/disable
Cache size	2, 4, 8, 16, 32, 64 (KB)
Cache line size	16, 32, 64 (B)
Cache ports	1, 2, 4, 8
Cache associativity	4, 8
Cache line flush	84 ns/line
Cache line invalidate	71 ns/line
Hardware prefetchers	Strided
MSHRs	16
Accelerator TLB size	8
TLB miss latency	200 ns
System bus width	32, 64 (b)

Figure 5.3: An example SoC that can be modeled using gem5-Aladdin. The table shows the set of design parameters that we swept and their values; this is just a small subset of what can be changed.

connected through the system bus. To understand how system-level effects impact the behavior of accelerators, we need simulation infrastructures that can model these heterogeneous systems. In this work, we integrate Aladdin with the gem5 system simulator[32], a widely-used system simulator with configurable CPUs and memory systems.

gem5-Aladdin models interactions between accelerators and CPUs, DMA, hardware-managed caches, and virtual memory. All of these features have implications on how the accelerator behaves and in the following sections, we describe how each is modeled.

5.3.1 Overview

We run gem5-Aladdin in syscall emulation mode, where it emulates system calls by passing it to the host operating system instead of simulating an operation system, because it is sufficient to capture the effects of our system-level considerations on performance and power. Full-system simulation would enable us to model operating system effects, but most are beyond the scope of this study. Some interactions with the operating system, such as device driver to hardware interactions, are important to our studies. These are characterized through real hardware measurements and analytically included in our models. Finally, syscall emulation is much faster than full system simulation, easing rapid design space exploration.

5.3.2 Accelerator Modeling

The Aladdin accelerator simulator[145] takes a first step towards modeling the power, performance, and cycle-level activity of standalone, fixed-function accelerators without needing to generate RTL. Aladdin is a trace-based accelerator simulator that profiles the dynamic execution of a program and constructs a dynamic data dependence graph (DDDG) as a dataflow representation of an accelerator. The vertices in the DDDG are LLVM IR instructions, and the edges represent true dependences between operations. Aladdin then applies common accelerator design optimizations and schedules the graph for execution through a breadth-first traversal, while accounting for user-defined hardware constraints. Aladdin was validated to be within 7% accuracy compared to standalone, RTL accelerator designs.

However, Aladdin only focuses on the standalone datapath and local memories. It assumes that all data has been pre-loaded into the local scratchpads. This skips the modeling of any interactions between accelerators and the rest of the system in which they belong.

5.3.3 DMA Engine

DMA is a software managed mechanism for transferring bulk data without CPU intervention. To set up a transaction, the programmer constructs a DMA transfer descriptor that contains the source and destination memory addresses along with the size of the transfer. Multiple descriptors can be constructed and connected through a linked list. When all descriptors are ready, the programmer initiates the transfer by writing the address of the head of the descriptor linked list into a hardware DMA engine's control register. The DMA engine then fetches and services these descriptors one by one. Meanwhile, the CPU is free to perform other work.

In gem5-Aladdin, accelerators can invoke the DMA engine already present in gem5. To do so, a programmer inserts calls to special `dmaLoad` and `dmaStore` inside the accelerated function with the appropriate source, destination, and size arguments. When the function is traced by Aladdin, Aladdin will identify these calls as DMA operations and issue the request to the gem5 DMA engine. As part of the DMA engine, we include an analytical model to account for cache flush and invalidation latency, using the measured numbers mentioned in Section 5.4.2.

5.3.4 Caches and Virtual Memory

For the accelerator caches, we use gem5's classic cache model along with a basic MOESI cache coherence protocol. When Aladdin sees a memory access that is mapped to a cache, it sends a request through a cache port to its local cache. Aladdin will receive a callback from the cache hierarchy when the request is completed. To support virtual memory, we implement a special Aladdin TLB model. We don't use gem5's existing TLB models for two reasons. First, the existing TLB models are tied to particular ISAs, which do not pertain to accelerators. Second, as a trace-driven simulator, the trace address that Aladdin originally uses does not directly map to the simulated address space that CPU is accessing. To maintain correct memory access behavior, our custom TLB model translates the *trace* address to a simulated *virtual* memory address and then to a simulated *physical* address. TLB misses and page table walks are modeled with a pre-characterized miss penalty.

5.3.5 CPU-Accelerator Interface

On the CPU, a simulated user program can invoke an attached accelerator through the `ioctl` system call, a system call widely used in practice for arbitrary communication with devices. In the `ioctl` emulation code, we assign a special file descriptor value for Aladdin and use command numbers to refer to individual accelerators. When the accelerator finishes, it writes to a shared pointer between the CPU and the accelerator. The CPU will see the update due to cache coherence. After invoking the accelerator, the CPU can either spin wait for the status to update or continue to do other work, periodically checking the status variable to see if the accelerator is completed.

5.3.6 Performance Validation

We validated gem5-Aladdin's performance models using the Zynq Zedboard for a subset of the MachSuite benchmark suite. For each benchmark, we implement the AXI4-Stream interface to transfer data via Xilinx's DMA IP blocks. Accelerator RTL is generated using Vivado HLS 2015.1. To maintain a consistent view of the model, we use HLS without

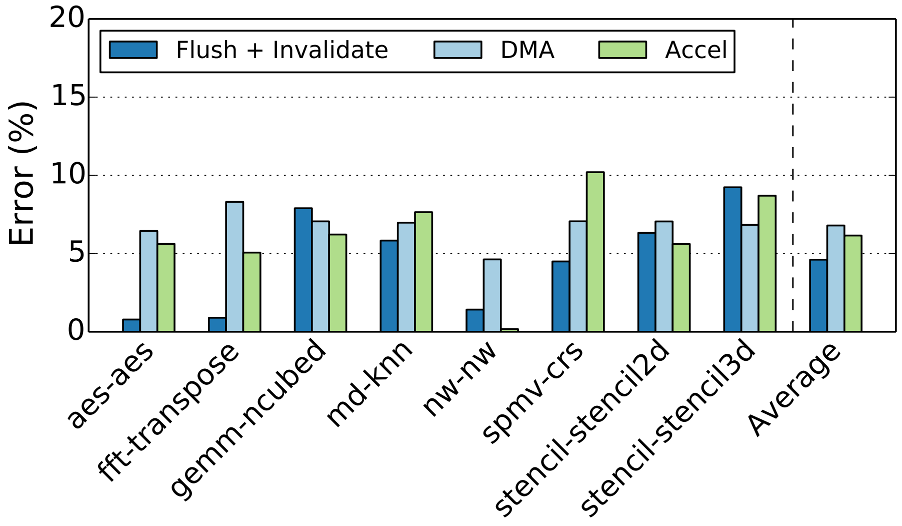


Figure 5.4: Error between Zedboard and gem5-Aladdin cycles.

specifying any additional design optimizations, so Vivado HLS generates a default design whose parameters we then match in Aladdin.

The complete system (including the DMA engine, accelerators, crossbars, etc.) is implemented in in Vivado Design Suite 2015.1. Software running on the CPU first initializes all devices in the system and generates the accelerator input data. Then it performs the necessary cache flushes and invalidates and starts the DMA transfer. The accelerator automatically begins computation when the DMA transfer is complete.

To measure performance, we instrument this code using cycle counters on the A9 CPUs. Because we cannot directly measure the DMA transfer time, we include logic analyzers in the synthesized system to capture waveforms using Xilinx tools during live execution. Most benchmarks were implemented on a 10ns clock; a few used slower clocks for timing reasons.

The results of our validation are shown in Figure 5.4. Our DMA performance model achieves 6% average error across this suite of benchmarks, while Aladdin achieves 7% average error, and the flush and invalidation analytical model achieve less than 5% average error. These results demonstrate the ability of gem5-Aladdin to model a wide range of accelerator workloads accurately for both the accelerated kernels and important system-level considerations.

Validation omissions

Our validation focuses on the features required by our DMA techniques: cache flushes and invalidates, DMA transfer time, and accelerator runtime. In general, we validated as much of the new additions as we could. Below are the components this work does not validate and our reasons for omitting them.

- CPU performance models: Existing work by Gutierrez et al. has already produced an accurate gem5 CPU model for the ARM A9 core[77], and gem5-Aladdin uses that validated model.
- Power model: All power results represent only the accelerator power. We do not account for CPU power in any of our results. We use the same validated Aladdin's power models with TSMC 40nm technology.
- Cache: To the best of our knowledge, there is no existing IP block available on Zynq such that we could implement a cache controller on the programmable fabric. Furthermore, we never modified the cache model of gem5.

5.4 Memory System Opportunities

In this section, we will discuss the primary design considerations when deciding whether to use a DMA- or cache-based memory system for an accelerator. Because baseline DMA leaves much room for improvement, we will also apply two optimizations to DMA. We will then describe design considerations specific to cache-based accelerators. Finally, we will evaluate the performance of both memory systems for a set of representative benchmarks.

5.4.1 Primary design considerations

First, we compare and contrast DMA and caches across the three system-level considerations mentioned earlier: push vs. pull, data movement granularity, management of coherency, and behavior under shared resource contention.

Push vs. Pull: DMA is designed for efficient bulk data transfer where the data requirements of the program are well known a priori. This works well for streaming applications and applications with high compute-to-memory ratios. However, applications with more irregular memory access patterns, such as indirect memory accesses, can suffer without an on-demand memory system such as a cache. In addition, because caches have the feature of automatic cache line replacement, a cache can often afford to be smaller than a scratchpad that must hold all the data.

Data Movement Granularity: Because DMA is software controlled, the overheads of setting up a transaction are usually amortized over a large bulk transfer. In contrast, caches pull in data at cache line granularity, enabling fine-grained overlap between compute and data movement. Although fine-grained DMA is possible, each new transactions adds additional overheads. On the other hand, caches must perform tag comparisons, replacements, and typically address translations, which make them inefficient for bulk data movement.

Cache Coherence Management: DMA engines typically can only access main memory or the last level cache. Therefore, the programmer must conservatively flush any data the accelerator may read out of private caches. Figure 5.2b shows that on average, accelerators employing traditional DMA spend 20% of their total cycles on cache flushes. The flush is typically performed by software because DMA engines rarely participate in coherency (although there have been exceptions, such as the IBM Cell[92]). In contrast, hardware-managed caches handle all of this complexity transparently at the cost of additional hardware.

Shared Resource Contention: In a real scenario where resources such as the main system interconnect and main memory are shared across multiple agents, invariably a DMA operation or cache fill will stall to allow another process to make progress. A coarse-grained mechanism such as DMA will be affected much more by shared resource contention because the accelerator usually waits for the entire transfer to complete. In comparison, fine-grained memory accesses such as cache fills are less likely to contend due to their smaller size, and hit-under-miss allows other independent operations to proceed even while an earlier cache load or store missed.

5.4.2 DMA Optimizations

In this section, we improve the baseline DMA method by overlapping various stages of the process. We examine two DMA latency optimizations, pipelined DMA and DMA-triggered computation, which are graphically depicted in Figure 5.5.

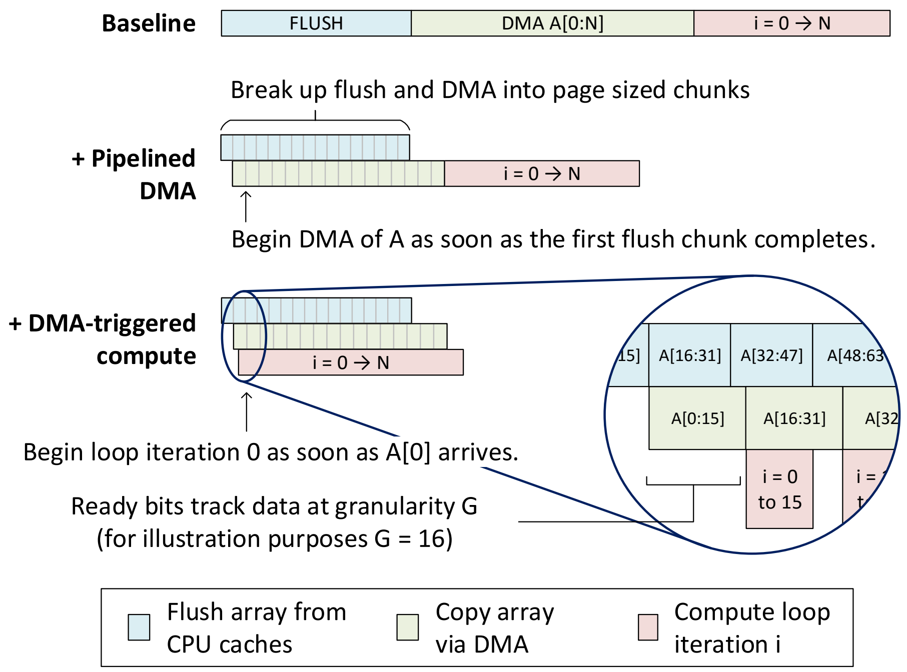

Baseline
FLUSH
DMA A[0:N]
i = 0 → N
Break up flush and DMA into page sized chunks
+ Pipelined DMA
i = 0 → N
Begin DMA of A as soon as the first flush chunk completes.
+ DMA-triggered compute
i = 0 → N
A[16:31]
A[32:47]
A[48:63
A[0:15]
A[16:31]
i = 0 to 15
Begin loop iteration 0 as soon as A[0] arrives.
Ready bits track data at granularity G (for illustration purposes G = 16)
Flush array from CPU caches
Copy array via DMA
Compute loop iteration i

Figure 5.5: A demonstration of the DMA latency reduction techniques, using `reduction` as an example workload.

Pipelined DMA

Pipelined DMA reduces latency by dividing the flush and DMA operations into page sized blocks and overlapping the DMA of block b with the flush of block $b+1$. We choose page size granularity to optimize for DRAM row buffer hits. In the best case, we can hide all but 4KB of the flush latency. Note that the correctness of this optimization is ensured by never starting a DMA block before its flush has completed.

Cache line flush latency varies across ISAs and implementations. For example, we characterized the flush throughput on the Zedboard's Cortex A9 CPU to be one cache line per 56 cycles at 667MHz. To achieve optimal pipelining and avoid bubbles, we want to match the flush and DMA latencies of a 4KB transaction. On the Zedboard, this is achieved with an accelerator clock frequency of 100MHz, which is why we use this frequency for the rest of our experiments.

Breaking up a large flush and DMA operation introduces additional overheads. The DMA engine must fetch new metadata from main memory for every block, andonly the

CPU must synchronize flushes with dependent DMA operations. For this, we add a fixed 40 cycle delay to every DMA transaction, also based on characterization. At 100MHz, this accounts for metadata reads (4 cycles), the one-way latency of initiating DMA from the CPU (17 cycles), and additional CPU cycles spent on housekeeping actions.

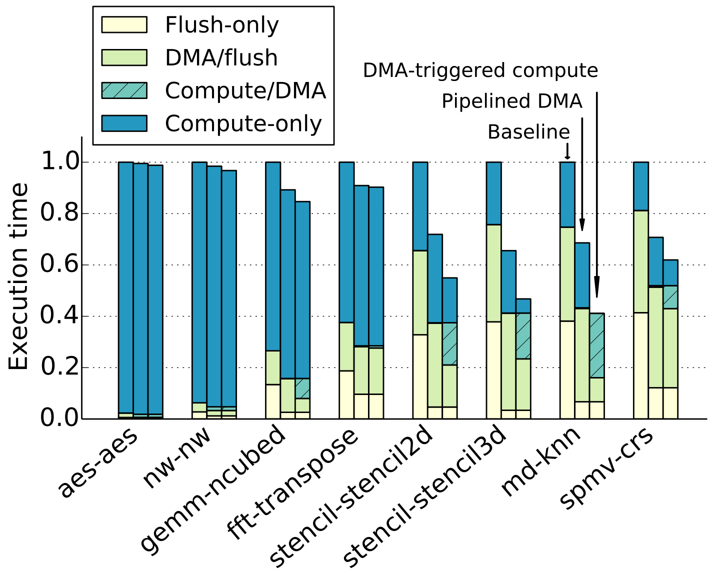
DMA-Triggered Computation

Even with pipelined DMA, the accelerator still must wait for the entire DMA transaction to finish before it can start. To overcome this, we augment our accelerators with full/empty-bits, which are often used in producer-consumer situations to indicate that data is ready[109]. In our designs, we track data at cache line granularity to be consistent with the preceding flush operations (which operate on cache lines). Full/empty bits are stored in a separate SRAM structure and indexed by a slice of the load address. With full/empty bits, the accelerator immediately begins computation without waiting for DMA to complete until it reaches a load. A load accesses both the full/empty bit arrays and the data arrays in parallel and returns the data if the full/empty bit is 1. If not, the control logic stalls the datapath until the DMA engine eventually fills that data and sets the full/empty bit. Note that double-buffering could be implemented in this scheme by tracking the granularity of data transfer at half the array size instead of cache line size, without any manual intervention. If an accelerator has multiple datapath lanes, other lanes are free to proceed even while some are blocked.

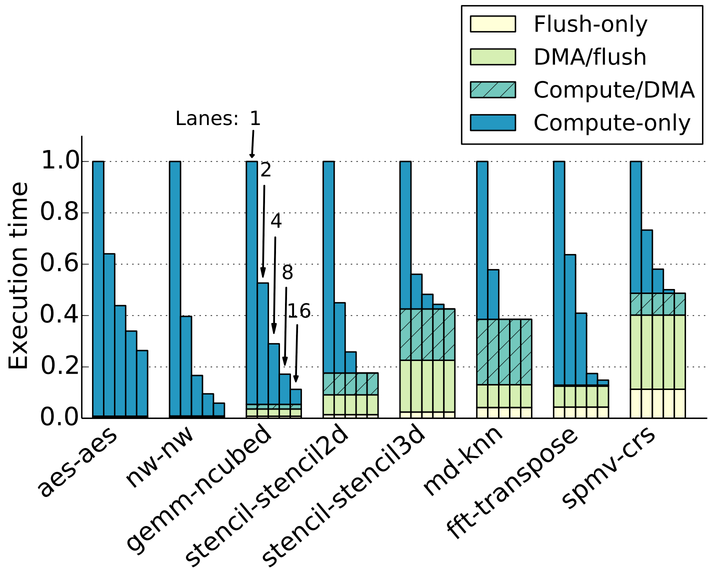
5.4.3 DMA Evaluation

To quantify the performance improvements from each of the techniques described, we start from the baseline design and cumulatively apply our DMA optimizations. From execution traces, we break down the runtime into four parts based on how cycles are spent: flush-only time, DMA/flush time, compute/DMA time, and compute-only time. Flush-only and compute-only are self-explanatory; compute/DMA time includes all cycles when compute and DMA are overlapped, while DMA/flush includes all cycles when DMA and flush *but not compute* are running.

Increasing the parallelism of accelerator datapaths through additional datapath lanes and memory partitioning is a widely used and effective way to achieve higher performance at the cost of greater area and power. However, the presence of memory movement imposes an upper bound on achieveable speedup, and our DMA optimizations will



(a) Performance improvements from each technique.



(b) Effect of parallelism on performance gains.

Figure 5.6: Cumulatively applying each technique reduces the additional cycles spent on DMA, with some benchmarks seeing more benefit than others. After applying all techniques, increasing parallelism through loop unrolling reduces compute cycles until near-complete overlap is achieved, causing performance to saturate.

affect realized speedup as well. To understand how parallel an accelerator must be in order to approach this upper bound, we take all the optimizations, sweep the parallelism of the accelerator datapath, and analyze the speedups realized.

Performance gains from DMA optimizations

The performance improvements from each optimization are shown in 5.6a. For brevity, we present only a subset of benchmarks whose DMA times spans the range shown in Figure 5.2b. We fix the parallelism of all accelerators to four datapath lanes.

We immediately observe that in the baseline design, flush-only time is a significant fraction of the total execution time. Pipelined DMA is thus shown to be effective, almost completely eliminating flush-only time for all the benchmarks shown. This is because the benefits of pipelined DMA are dependent only on the amount of data transferred and not on the memory characteristics of the application.

DMA-triggered computation is able to improve performance even more, but its effectiveness clearly varies across workloads. It is most effective when a benchmark exhibits some level of streaming behavior. For example, `stencil2d` uses a 3x3 kernel and thus requires only the first three rows of the input matrix to arrive before it can start computation, so ready bits recover a significant amount of performance. A similar logic applies to `md-knn` – in fact, ready bits are so effective here that with just four datapath lanes, we achieve 99% compute/DMA overlap. This is in contrast to `fft-transpose`, where each unit of work requires eight loads strided across the entire input arrays. This is not a streaming memory access pattern and so DMA-triggered compute is ineffective.

Impact of parallelism on DMA optimizations

The results of sweeping accelerator parallelism, while applying all the DMA optimizations, is shown in Figure 5.6b. This figure demonstrates two points.

First, with enough parallelism, on several workloads, the entire computation can be overlapped with DMA. This means that without reducing flush or DMA time, no more speedup is achievable. Benchmarks without this property either have little data to transfer to begin with (`aes`) or are so serial that they don't benefit from data parallelism in the first place (`nw`).

Second, increased parallelism has no effect on the amount of compute-DMA overlap. This is due to the *serial data arrival* effect: no matter how parallel a datapath is, DMA will always copy data sequentially starting with the first byte, and until that critical first byte of data arrives, no compute can start. As our DMA engine already fully utilizes the available bus bandwidth, this data cannot arrive any faster, and therefore compute also cannot be overlapped any more.

In conclusion, these sweeps show that memory movement, not compute, has become a significant bottleneck, and accelerating computation only will quickly bring diminishing returns. In fact, Figure 5.6b shows that for many benchmarks, we can achieve the upper bound performance with relatively few datapath lanes! As a result, to continue to get better performance, we must somehow further overlap computation with data by overcoming the serial data arrival effect, motivating the study of fine-grained, *on-demand* memory systems.

5.4.4 Cache-Based Accelerators

In a cache-based accelerator, one of the most important questions is how to handle variable latency memory accesses in a statically scheduled datapath. The simplest way is to stall the entire datapath until the miss resolves, but this significantly hurts performance. Techniques such as multithreaded accelerators have been proposed in the CAD community to hide cache miss latency[114,85], but these require additional resources to store thread contexts.

We choose a simpler cache miss handling scheme. Accelerators are typically designed with multiple parallel lanes, When a cache miss happens in one of the lanes, only that lane is stalled until the miss resolves. Other lanes are free to continue. We include MSHRs to enable hit-under-miss and multiple outstanding misses. Any lane with a dependence on a result from a blocked lane is also blocked via control logic mechanisms. This scheme lets independent computation proceed while waiting for the missed data to be returned without requiring storage for thread contexts. When lanes are finished executing, they must wait and synchronize with all other lanes before the next iteration can begin.

Another important design choice is what data is cached. In our experiments, only data that must be eventually shared with the rest of the system is sent through the cache, and local scratchpads are used for private intermediate data. For example, `nw` uses an internal score matrix to align DNA sequences. This matrix is kept in local scratchpads.

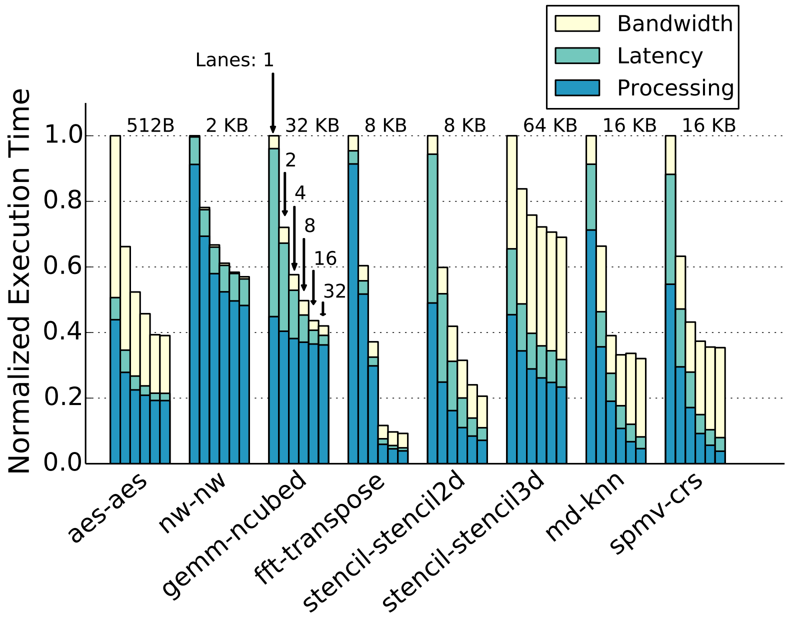


Figure 5.7: Effect of datapath parallelism on cache-based accelerator performance.

5.4.5 Cache Evaluation

Next we analyze the impact of datapath parallelism on cache-based accelerator performance. We decompose total execution time into processing time, latency time, and memory bandwidth time, using a similar technique as Burger et al.[37]. Each component is the additional execution time after applying a realistic constraint to a memory system parameter. To briefly summarize:

1. Processing time: assume memory accesses are single-cycle and always hit.
2. Latency time: allow memory accesses to miss in the cache, but the system bus has unlimited bandwidth to service cache fills.
3. Bandwidth time: constrain the system bus width to 32 bits, thus limiting the rate at which cache fill requests can be serviced.

Impact of Datapath Parallelism

Figure 5.7 shows how the performance of cache-based accelerators scales with datapath parallelism. In this set of experiments, we first sweep cache sizes to find the smallest cache at which performance saturates for each benchmark. This is labeled at the top of each group of bars. The datapath parallelism sweep is performed with this cache size per benchmark.

Naturally, we observe that processing time decreases with increased parallelism, as expected. However, parallelism also improves *latency* time, which is in contrast to the DMA experiments where parallelism did not affect flush or DMA time. This is because caches are a fine-grained pull-based memory system, and increased datapath parallelism also increases memory-level parallelism (more memory accesses per cycle). Furthermore, the fine granularity more effectively masks cache miss latency with computation, thereby decreasing latency time.

On the other hand, more parallelism does not improve bandwidth time due to increased memory bandwidth pressure. In fact, bandwidth time becomes a larger fraction of total execution time as we go to increasingly parallel designs. For example, the performance of `spmv-crs` and `md-knn` is eventually bottlenecked by bandwidth, even though the increased memory level parallelism improves both processing and latency time. Accelerators that are designed without consideration of the available memory bandwidth in the SoC are likely to be over-designed, provisioning more functional units than can be fed with data by the system.

5.5 Accelerator Design Choices

Thus far, we have discussed in detail how the performance of accelerated workloads changes when connected to two different memory systems, scratchpad with DMA and hardware-managed caches. However, it has been unclear when to select one over the other. Performance is not the only goal as well; accelerator designers especially must balance performance targets against power and energy constraints. It is also unclear how differently one must think about designing accelerators when system-level effects such as data movement and its mechanisms are considered. In this section, we shed light on the DMA vs. cache question as well as illustrate that without consideration for data movement, accelerator designers are highly to overprovision and underutilize their accelerators.

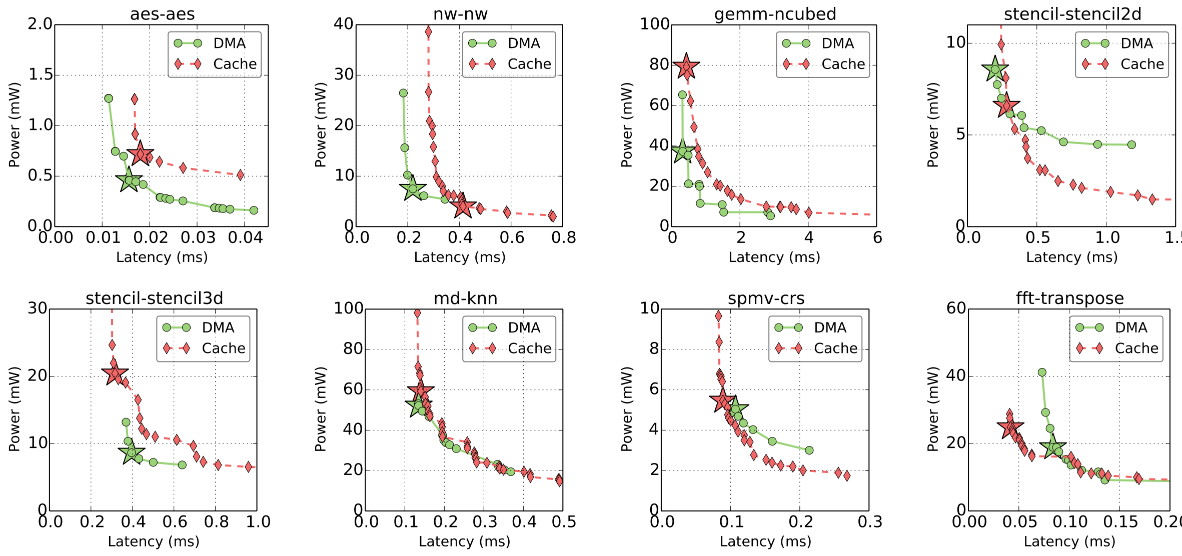


Figure 5.8: Power-performance Pareto curves for DMA- and cache-based accelerators. EDP optimal design points are shown as stars. Benchmarks are ordered left-to-right, top-down by preference for a DMA-based vs. a cache-based memory system.

5.5.1 DMA vs. Caches

One of the earliest decisions a designer needs to make is decide whether private scratchpads with DMA or hardware-managed caches is a better fit for the application at hand. In this experiment, we performed a comprehensive design space sweep for all the parameters listed in Figure 5.3 for all of the MachSuite benchmarks. We show the resulting Pareto optimal design curves, distinguished by memory system type, in Figure 5.8. For brevity, we show only eight benchmarks that span the range of design space characteristics observed. The energy-delay-product (EDP) optimal design point for each memory system is labeled with a star of the corresponding color. All DMA design points apply all the optimizations discussed in Section 5.4.2.

This experiment shows that some benchmarks umambiguously prefer scratchpads with DMA (on the left), some clearly are better with caches (on the right), and several work equally well with either (in the middle). We will briefly discuss each benchmark's behavior in turn.

`AES-AES` AND `NW-NW` These two benchmarks always both perform better and use less power with DMA than with caches. They have have regular access patterns, and importantly, they require only a small amount of data before computation can be triggered. In contrast, a cache-based memory system will first experience a TLB miss followed by cache misses, causing significant performance slowdown.

`GEMM-NCUBED` This benchmark, unlike the previous two shows the cache design matching its DMA counterpart in performance. However, due to the various overheads of caches (tag lookups, TLB lookups, etc.), more power must be expended to reach this performance.

`STENCIL-STENCIL2D` Although DMA can always outperform the cache system on this benchmark, a cache-based design achieves comparable performance with lower power, because the cache system captures enough locality to use a *smaller* cache, whereas the scratchpad design must fit the entire data set into local memory.

`STENCIL-STENCIL3D` The 3D stencil kernel distinguishes itself from its 2D counterpart because the cache system can outperform the optimized DMA system at the cost of additional power. This is because the kernel's three-dimensional memory access pattern creates nonuniform stride lengths, which are gracefully handled by the on-demand nature of a cache. In contrast, even the most optimized DMA design spends half of its execution time waiting for DMA and flush operations. The cost of this performance is $2\times$ to $3\times$ increased power.

`MD-KNN` `md-knn` is a compute intensive application. In this benchmark, there are 12 FP multiplies per atom-to-atom interaction, so the power consumption of this benchmark is dominated by functional units rather than memory. Also, the optimized DMA system is able to fully overlap compute with data movement because full/empty bits are effective in this benchmark. Figure 5.8 shows that the Pareto curves for cache and DMA designs largely overlap, demonstrating that either memory system can be an acceptable choice.

`SPMV-CRS` On this benchmark, a cache system is able to outperform a DMA system with lower power as well. This is due to the indirect memory accesses inherent to sparse matrix multiply algorithms, where the first set of loads provide the memory addresses for the next set that actually returns the data. Full/empty bits may not be effective on this benchmark if the data pointed to by a matrix index has not yet arrived, since DMA sends data sequentially, but a cache can fetch arbitrary memory locations. Caches thus eliminate most of the idling time, leading to better performance. Lower power on caches is achieved by being able to use a smaller cache than the scratchpads.

FFT-TRANSPOSE `fft-transpose` also performs better with caches than with DMA but for slightly different reasons. There are no indirect memory accesses in this benchmark. Instead, the parallel implementation of this benchmark possesses a stride length of 512 bytes, meaning that each loop iteration (aka datapath lane) reads only eight bytes per 512 bytes of data. As a result, even with full/empty bits, a DMA system must supply nearly all of the data before the computation can begin, whereas this is not a problem for the cache system. Again, lower power is achieved by a smaller cache than scratchpads.

5.5.2 Design Decision Comparison

In addition to deciding the type of memory system to use, accelerator designers must also select several local parameters, such as the datapath parallelism and local memory size and bandwidth. These parameters are central to performance and energy efficiency, and in this section, we show that when system-level effects are considered, these parameters can change considerably compared to when an accelerator is designed in isolation.

We consider the following design scenarios to illustrate the optimal designs of accelerators are affected by system-level effects:

1. Baseline: design accelerators in isolation.
2. Co-designed DMA: use DMA to transport data over a 32-bit system bus.
3. Co-designed cache: use a hardware-managed cache for the accelerator's local memory.
4. Co-designed cache with 64-bit bus: Same as above, but we double the width of the system bus.

We focus our comparisons on three accelerator microarchitectural parameters: datapath lanes, local SRAM size, and local memory bandwidth to datapath lanes. As before, we select the EDP optimal points from each design scenario for comparison.

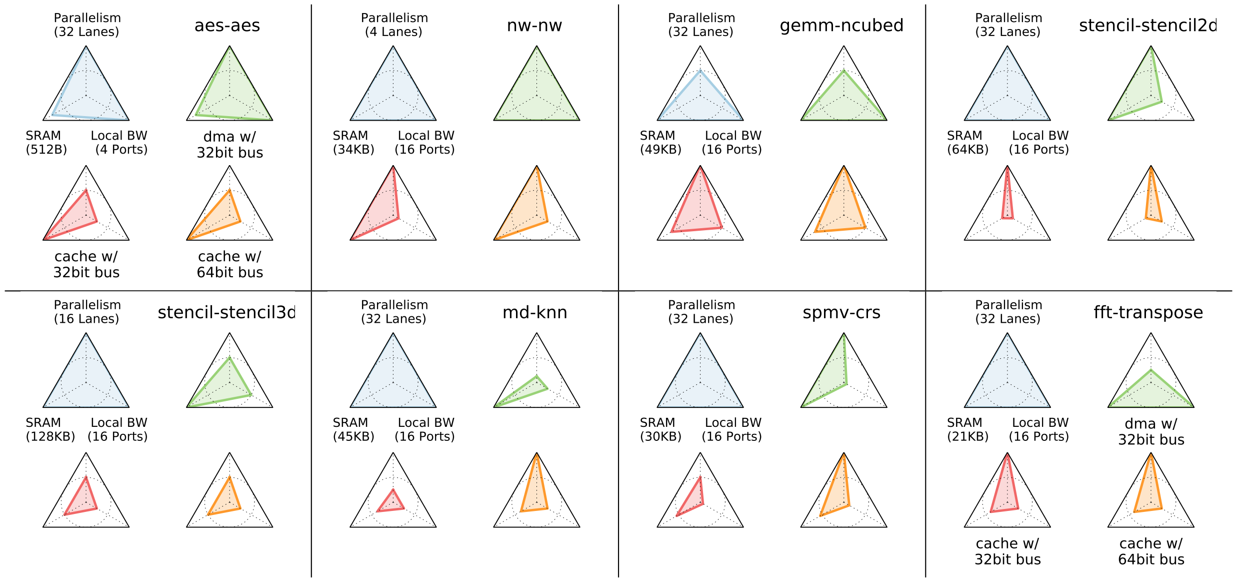


Figure 5.9: Comparison of accelerator microarchitectural parameters across four design scenarios. The vertices of the kiviat plots represent the number of datapath lanes, SRAM sizes, and local memory bandwidth, normalized to the maximum across the four scenarios for each benchmark.

Figure 5.9 shows the differences in these three dimensions in the aforementioned scenarios for each benchmark. Each set of four triangles represents the optimal designs of an accelerator for a benchmark under the four scenarios. For visual clarity, only the first and last groups include scenario legends. For each benchmark, we normalize each axis to the maximum value across all scenarios, indicated in parentheses. The following subsections discuss the differences in designs.

Isolated vs Co-Designed Microarchitecture

It is immediately apparent that accelerators designed in isolation overprovision accelerator resources. As Figure 5.9 shows, isolated designs tend to pick the largest design values across all four scenarios. In isolation, this makes sense, but when system-level considerations are accounted for, we find a more balanced design that provisions fewer computational resources.

This overdesign is most pronounced in local memory bandwidth and SRAM size for cache-based designs. Isolated designs attempt to parallalize computation as much as

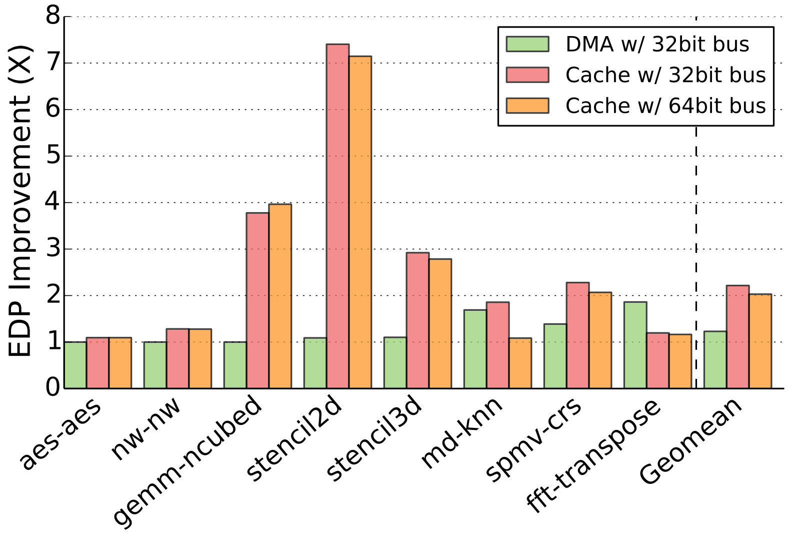


Figure 5.10: Energy delay product improvement of co-designed accelerators in different scenarios compared to the isolated designs. The design parameters of each optimal design points are illustrated in Figure 5.9.

possible, requiring high internal memory bandwidth, but in a more realistic environment, the need to move data from system to accelerator imposes a upper bound on performance that makes internal memory-level parallelism less critical. For example, on `spmv-crs` and `md-knn`, both DMA- and cache-based designs require much lower local memory bandwidth than the isolated design. In addition, because caches have the feature of automatic data replacement, they can be sized smaller than scratchpads which must hold all the data, resulting in energy improvements.

In general, caches tend to prefer more parallel datapaths than DMA, as shown in `md-knn` and `fft-transpose`, since their fine-grained nature allows more parallel memory accesses. In fact, `gemm-ncubed` an example where a co-designed cache-based accelerator is more parallel than both the isolated design and a DMA-based one.

Impact of System Bus Bandwidth

As a proxy for system contention in loaded system, we vary system bus width to modulate the bus bandwidth available to accelerators. If we compare accelerators designed with a 64-bit bus to those designed with a 32-bit bus (bottom two triangles in Figure 5.9), we see that accelerators designed with lower bus bandwidth tend to be both less datapath parallel (`md-knn`, `spmv-crs`) and provision less local memory bandwidth (`nw`, `stencil2d`, and `spmv-crs`). These effects happen for the same reasons co-designed accelerators are leaner than isolated accelerators.

EDP Improvement

Figure 5.10 shows the improvements in EDP when accelerators are co-designed, compared to how an accelerator designed in isolation would behave under a more realistic system. This is the same analysis as Figure 5.1, but applied to more benchmarks and three different design scenarios. Overall, average EDP improves by 1.2×, 2.2×, and 2.0× for accelerators with DMA, caches with 32-bit system bus, and caches with a 64-bit bus, respectively.

The EDP improvements for co-designed cache-based accelerators is higher than that for DMA-based accelerators because an overly aggressive design for a cache results in large, highly multiported cache, and this is much more expensive to implement than a partitioned scratchpad. Furthermore, we see that the improvements are greater when we co-design with a 32-bit bus compared to a 64-bit bus. In other words, co-design is even more important for contended systems than uncontended systems.

5.6 Related Work

There has been much prior work on integrating private local memories of accelerators with CPUs and the rest of the system. We have discussed accelerator-system codesign, caches, coherency, and virtual memory for accelerators, DMA optimizations, simulation frameworks and prototyping platforms, and we will discuss how this work relates to and differs from past work.

The explosion of interest in hardware accelerators has spurred a great deal of creativity in designing efficient circuits for specific workloads. However, a lot of this work focuses on how to optimize the computation performed, ignoring how the data the computation requires will arrive.

We observe that there are two classes of accelerated workloads for which optimization of data movement from global memory to local memory is absolutely critical: big data applications and near-data processing applications. Accelerators for `memcached`[103], database partitioning[164], and those built with near-data CGRAs[67] all contain specialized interfaces codesigned with the system bus interface for efficient communicate with the rest of the system, or are tightly coupled accelerators that rely on the existing general purpose core for data, e.g., Convolution Engine[129].

Caches, coherency, and virtual memory have been well studied in the GPU literature[79,21,125,127,124]. Only recently has there been movement in this field for more fixed function, less programmable accelerators, such as IBM's Coherent Accelerator Processor Interface[152], AXI Accelerator Coherency Port[119], and the Intel HARP platform[36]. There has also been some work on specialized coherence protocols for accelerators, such as Fusion[96], and hybrid memory models for heterogeneous platforms[89]. With access to global memory spaces, protecting the SoC and accelerators from unsafe memory accesses has also been investigated[121]. The IBM Cell BE architecture featured a hardware coherent DMA engine, which addresses some of the issues we have raised[42].

Finally, others have integrated accelerator models with gem5, such as gem5-gpu[126] and PARADE[53]. PARADE is similar to gem5-Aladdin in that it enables simulation of accelerated workloads with an SoC framework. However, it only models the traditional DMA-based accelerators where all data must be copied to local scratchpads before compute begins. In contrast, gem5-Aladdin is able to model a cache-based accelerator with variable latency memory accesses as well as various optimizations on DMA to reduce idle time. The DMA engine must fetch new metadata from main memory for every block, and

"If your plan is for one year, plant grain. If your plan is for ten years, plant trees. If your plan is for one hundred years, educate children."

Guan Zhong, Spring and Autumn Period.

6 Conclusions and Future Directions

Specialized architectures have been a growing topic in both academic research and commercial development for the past decade. As traditional technology scaling slows, specialization becomes a viable solution for computer architects to continue performance growth and energy efficiency improvements without relying on technological advances.

This dissertation is one of the first efforts in the computer architecture community to abstract the traditional, RTL-based design methodology into the architectural level. We address three key issues that arise in design and modeling for specialized architectures. First, we developed an ISA-independent workload characterization methodology to characterize workload intrinsic characteristics, without the artifacts from micro-architecture and ISA features. Second, the lack of simulation infrastructures in accelerator design motivated us to the development of Aladdin, a pre-RTL, power, performance, and area simulator for accelerators. We rigorously validated Aladdin against RTL implementations and showed that Aladdin can accurately model accelerator power, performance and area compared to accelerator designs generated by traditional RTL flows. In addition, Aladdin provides these estimates over 100× faster. Third, we integrated Aladdin with the gem5 system simulator so that the integrated framework is able to capture dynamic interactions between accelerators and the SoC platform, enabling new architectural research directions in future heterogeneous systems.

Today's mobile SoCs provide a starting point for thinking about future architectures, but many challenges must be addressed to make specialization more pervasive and cost-effective. New architectures, design tools, and programming paradigms will be required to make this approach pervasive. The approaches described in this dissertation demonstrate some initial steps towards design and modeling specialized architecture at the architectural level. A number of challenges remain to be addressed in future work. Here we highlight three major challenges in the field of specialized architectures as our community moves forward:

1. Flexibility. Specialized accelerators, especially fixed-function accelerators, are designed only for specific applications or domains of applications. *How should we choose a combination of different accelerable kernels to achieve a good balance between application coverage and energy efficiency?* Composing accelerators and general-purpose cores can address this issue if we can understand the common kernels across a group of applications and identify efficient communication channels to chain these kernels so that they can work together to achieve greater functionality.

2. Design Cost. The increasing volume and diversity of accelerators in every generation of processors requires rolling out new designs quickly with relatively low design cost. RTL-based implementations through standard flows are inherently tedious and time-consuming. High-level synthesis tools have shown promise to speedup the design process. However, existing tools still face challenges to generate high-quality designs within reasonable time. *How can we rapidly create and validate new accelerators with good quality of design?* To achieve this, we must develop new program representations, compilation heuristics, and algorithms that can aid high-level synthesis tools to quickly generate high-quality designs. Domain-specific high-level synthesis approaches also are a promising direction.

3. Programmability. Programming modern high-performance SoC is similar to the state of GPGPU programming before the introduction of language extensions such as CUDA and OpenCL. Prior to the introduction of those languages, leveraging GPUs for general-purpose computing was possible only for experienced high-performance computing or game programmers. Similarly, the sophisticated features of today's SoCs are encapsulated in high-level library interfaces written by embedded-system programmers with detailed knowledge of the underlying SoC architecture. However, each generation of SoC requires a new software engineering

effort due to the development of new accelerators, local and shared memories, and communication interfaces. As we build more accelerators in future SoCs, we need to answer *what kinds of programming interfaces can we give to programmers* and *what architectural improvements can make programming easier.* Therefore, there needs to be a conjoined effort at both the hardware and software layers to identify what information should be communicated across layers and how we can design hardware better to improve its programmability.

References

[1] Altera SDK for OpenCL. http://www.altera.com/products/software/opencl/opencl-index.html.

[2] Are 28nm Transistors the Cheapest...Forever? https://www.semiwiki.com/forum/content/2768-28nm-transistors-cheapest-forever.html.

[3] Avago Agrees to Buy Broadcom for $37 Billion. http://www.wsj.com/articles/avago-to-buy-broadcom-for-37-billion-1432811311.

[4] Chipworks Disassembles Apple's A8 SoC: GX6450, 4MB L3 Cache & More. http://www.anandtech.com/show/8562/chipworks-a8.

[5] Coherent Accelerator Processor Interface (CAPI) for POWER8 Systems. IBM White Paper, September 2014.

[6] GCC4CLI. http://gcc.gnu.org/projects/cli.html.

[7] IBM Dumps Chip Unit and Pays GlobalFoundries $1.5 billion to Take the Business off its Hands. http://www.extremetech.com/computing/192430-ibm-dumps-chip-unit-pays-globalfoundries-1-5-billion-to-take-the-business-off-

[8] Intel Completes Acquisition of Altera. https://newsroom.intel.com/news-releases/intel-completes-acquisition-of-altera/.

[9] Intel Historical Development Cadence. http://www.anandtech.com/show/9447/intel-10nm-and-kaby-lake.

[10] Intel's CEO Brian Krzanich on Q2 2015 Earnings Call. http://seekingalpha.com/article/3329035-intels-intc-ceo-brian-krzanich-on-q2-2015-results-earnings-call-transcript.

[11] Intel® 64 and IA-32 Architectures Software Developer's Manual. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[12] LLVM assembly language reference manual, bitcode documentation. http://llvm.org/docs/LangRef.html.

[13] MathWorks HDL Coder. http://www.mathworks.com/products/hdl-coder/.

[14] Mckinsey on semiconductors: Moore's law: Repeal or renewal?

[15] Oracle's SPARC T4 Server Architecture. Oracle White Paper, June 2012.

[16] RISC-V Rocket Core. https://github.com/ucb-bar/rocket.

[17] Scala Programming Language. http://www.scala-lang.org/.

[18] The International Technology Roadmap for Semiconductors (ITRS). http://www.itrs.net/.

[19] TI OMAP Applications Processors. http://www.ti.com/product/omap5432.

[20] Xilinx Vivado High-Level Synthesis. http://www.xilinx.com/products/design-tools/vivado/.

[21] AMD. Compute Cores. www.amd.com/computecores, 2014.

[22] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[23] Wolfgang Arden, Michel Brillouët, Patrick Cogez, Mart Graef, Bert Huizing, and Reinhard Mahnkopf. More than moore white paper. 2:14, 2010.

[24] Joshua Auerbach, David F. Bacon, Ioana Burcea, Perry Cheng, Stephen J. Fink, Rodric Rabbah, and Sunil Shukla. A compiler and runtime for heterogeneous computing. In *Proceedings of the 49th Annual Design Automation Conference*, 2012.

[25] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: A java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, 2010.

[26] Todd M. Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer*, 2002.

[27] Todd M. Austin and Gurindar S. Sohi. Dynamic dependency analysis of ordinary programs. In *ISCA*, 1992.

[28] J. Bachrach, Huy Vo, B. Richards, Yunsup Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. Chisel: Constructing hardware in a scala embedded language. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, 2012.

[29] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *ISPASS*, 2009.

[30] Bilel Belhadj, Antoine Joubert, Zheng Li, Rodolphe Héliot, and Olivier Temam. Continuous real-world inputs can open up alternative accelerator designs. In *ISCA*, 2013.

[31] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[32] Nathan L. Binkert, Bradford M. Beckmann, Gabriel Black, Steven K. Reinhardt, Ali G. Saidi, Arkaprava Basu, Joel Hestness, Derek Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Computer Architecture News*, 2011.

[33] B. Blaner, B. Abali, B.M. Bass, S. Chari, R. Kalla, S. Kunkel, K. Lauricella, R. Leavens, J.J. Reilly, and P.A. Sandon. IBM POWER7+ processor on-chip accelerators for cryptography and active memory expansion. *IBM Journal of Research and Development*, 2013.

[34] Mark Bohr. A 30 year retrospective on dennard's mosfet scaling paper. *Solid-State Circuits Society Newsletter, IEEE*, 2007.

[35] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA*, 2000.

[36] Diane Bryant. Disrupting the data center to create the digital services economy. *Intel Announcement*, 2014.

[37] Doug Burger, James R. Goodman, and Alain Kagi. Memory bandwidth limitations of future microprocessors. In *ISCA*, 1996.

[38] Simone Campanoni, Giovanni Agosta, Stefano Crespi-Reghizzi, and Andrea Di Biagio. A highly flexible, parallel virtual machine: Design and experience of ildjit. *Software Practice Expererience*, 2010.

[39] Trevor E Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 52. ACM, 2011.

[40] Jared Casper and Kunle Olukotun. Hardware acceleration of database operations. In *FPGA*, 2014.

[41] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, 2009.

[42] T. Chen, R. Raghavan, J.N. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation - a performance view. In *IBM Journal of Research and Development*, 2007.

[43] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ASPLOS*, 2014.

[44] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Teman. Dadiannao: A machine-learning supercomputer. In *MICRO*, 2014.

[45] Chen, Yu-Hsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. In *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*, pages 262–263, 2016.

[46] Andrew A Chien. 10x10 must replace 90/10. In *Proceedings of the Salishan Conference on High Performance Computing*, 2010.

[47] Andrew A. Chien, Dilip Vasudevan, Tung Thanh Hoang, Yuanwei Fang, and Amirali Shambayati. 10x10: A case study in federated heterogeneous architecture. IEEE Micro, 2015.

[48] Eric S. Chung, John D. Davis, and Jaewon Lee. Linqits: big data on little clients. *ISCA*, 2013.

[49] Eric S. Chung, Peter A. Milder, James C. Hoe, and Ken Mai. Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus? In *MICRO*, 2010.

[50] Nathan Clark, Amir Hormati, and Scott A. Mahlke. VEAL: Virtualized Execution Accelerator for Loops. In *ISCA*, 2008.

[51] J. Cong, M.A. Ghodrat, M. Gill, B. Grigorian, Hui Huang, and G. Reinman. Composable accelerator-rich microprocessor enhanced for adaptivity and longevity. In *ISLPED*, 2013.

[52] Jason Cong, Yiping Fan, Guoling Han, and Zhiru Zhang. Application-specific instruction generation for configurable processor architectures. In *FPGA*, 2004.

[53] Jason Cong, Zhenman Fang, Michael Gill, and Glenn Reinman. Parade: A cycle-accurate full-system simulation platform for accelerator-rich architectural design and exploration. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 380–387. IEEE Press, 2015.

[54] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. Charm: a composable heterogeneous accelerator-rich microprocessor. In *ISLPED*, 2012.

[55] Jason Cong, Karthik Gururaj, and Guoling Han. Synthesis of reconfigurable high-performance multicore systems. In *FPGA*, 2009.

[56] Katherine Coons, Warren Hunt, Bertrand A. Maher, Doug Burger, and Kathryn S. McKinley. Optimal huffman tree-height reduction for instruction-level parallelism.

Technical Report TR-08-34, Department of Computer Sciences, The University of Texas at Austin, 2008.

[57] Paolo D'Alberto, Peter A. Milder, Aliaksei Sandryhaila, Franz Franchetti, James C. Hoe, José M. F. Moura, Markus Püschel, and Jeremy Johnson. Generating fpga accelerated DFT libraries. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2007.

[58] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, 2010.

[59] Andrew Danowitz, Kyle Kelley, James Mao, John P Stevenson, and Mark Horowitz. Cpu db: recording microprocessor history. *Communications of the ACM*, 2012.

[60] Robert H. Dennard, Fritz H. Gaensslen, Hwa-Nien Yu, V. Leo Rideout, Ernest Bassous, and Andre R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 1974.

[61] Lieven Eeckhout, Hans Vandierendonck, and Koen De Bosschere. Quantifying the impact of input data sets on program behavior and its applications. *Journal of Instruction-Level Parallelism*, 2003.

[62] Eldridge, Schuyler and Appavoo, Jonathan and Joshi, Ajay and Waterland, Amos and Seltzer, Margo. Towards General-Purpose Neural Network Computing. In *PACT*, 2015.

[63] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. *Micro, IEEE*, 2012.

[64] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, 2011.

[65] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In *MICRO*, 2012.

[66] Carlos Flores Fajardo, Zhen Fang, Ravi Iyer, German Fabila Garcia, Seung Eun Lee, and Li Zhao. Buffer-integrated-cache: a cost-effective sram architecture for handheld and embedded platforms. In *DAC*, 2011.

[67] Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim. Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules. In *High Performance Computer Architecture (HPCA)*, 2015.

[68] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. In *Symposium on Programming*, 1984.

[69] Brian A. Fields, Rastislav Bodík, and Mark D. Hill. Slack: Maximizing performance under technological constraints. In *ISCA*, 2002.

[70] Michael Fingeroff. *High-Level Synthesis Blue Book*. 2010.

[71] Karthik Ganesan, Lizy John, Valentina Salapura, and James Sexton. A performance counter based workload characterization on Blue Gene/P. In *International Conference on Parallel Processing*, 2008.

[72] Saturnino Garcia, Donghwan Jeon, Christopher M. Louie, and Michael Bedford Taylor. Kremlin: rethinking and rebooting gprof for the multicore age. In *PLDI*, 2011.

[73] Nithin George, HyoukJoong Lee, David Novo, Tiark Rompf, Kevin J Brown, Arvind K Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. Hardware system synthesis from domain-specific languages. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, 2014.

[74] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *HPCA*, 2011.

[75] Venkatraman Govindaraju, Tony Nowatzki, and Karthikeyan Sankaralingam. Breaking simd shackles with an exposed flexible microarchitecture and the access execute pdg. In *PACT*, 2013.

[76] Shantanu Gupta, Shuguang Feng, Amin Ansari, Scott Mahlke, and David August. Bundled execution of recurring traces for energy-efficient general purpose processing. In *MICRO*, 2011.

[77] Anthony Gutierrez, Joseph Pusdesris, Ronald G. Dreslinski, Trevor Mudge, Chander Sudanthi, Christopher D. Emmons, Mitchell Hayenga, and Nigel Paver. Sources of Error in Full-System Simulation. In *International Symposium on Performance Analysis of Systems and Software*, 2014.

[78] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. In *ISCA*, 2010.

[79] Mark Harris. Unified memory in cuda 6, 2013.

[80] A. Hartstein and Thomas R. Puzak. Optimum power/performance pipeline depth. In *MICRO*, 2003.

[81] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. Darkroom: Compiling high-level image processing code into hardware pipelines. In *SIGGRAPH*, 2014.

[82] Kenneth Hoste and Lieven Eeckhout. Comparing benchmarks using key microarchitecture-independent characteristics. In *International Symposium on Workload Characterization*, 2006.

[83] MS Hrishikesh, Doug Burger, Stephen W Keckler, Premkishore Shivakumar, Norman P Jouppi, and Keith I Farkas. The optimal logic depth per pipeline stage is 6 to 8 fo4 inverter delays. In *ISCA*, page 0014, 2002.

[84] Hillery C. Hunter and Wen mei W. Hwu. Code coverage and input variability: effects on architecture and compiler research. In *CASES*, 2002.

[85] Jens Huthmann, Julian Oppermann, and Andreas Koch. Automatic high-level synthesis of multi-threaded hardware accelerators. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–4. IEEE, 2014.

[86] Tensilica Inc. How to minimize energy consumption while maximizing asic and soc performance.

[87] Donghwan Jeon, Saturnino Garcia, Christopher M. Louie, and Michael Bedford Taylor. Kismet: parallel speedup estimates for serial programs. In *OOPSLA*, 2011.

[88] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *International Symposium on Computer Architecture (ISCA)*, 2015.

[89] John H Kelm, Daniel R Johnson, William Tuohy, Steven S Lumetta, and Sanjay J Patel. Cohesion: a hybrid memory model for accelerators. In *International Symposium on Computer Architecture*, 2010.

[90] R.E. Kessler. The Cavium 32 Core OCTEON II 68xx. *Hop Chips*, 2011.

[91] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. Sd3: A scalable approach to dynamic data-dependence profiling. In *MICRO*, 2010.

[92] Michael Kistler, Michael Perrone, and Fabrizio Petrini. Cell multiprocessor communication network: Built for speed. In *IEEE Micro*, 2006.

[93] David Koeplinger, Raghu Prabhakar, Yaqi Zhang, Christina Delimitrou, Christos Kozyrakis, and Kunle Olukotun. Automatic Generation of Efficient Accelerators for Reconfigurable Hardware. In *ISCA*, 2016.

[94] Anil Krishna, Timothy Heil, Nicholas Lindberg, Farnaz Toussi, and Steven VanderWiel. Hardware acceleration in the ibm poweren processor: Architecture and performance. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, 2012.

[95] Manoj Kumar. Measuring parallelism in computation-intensive scientific/engineering applications. *IEEE Trans. Computers*, 1988.

[96] Snehasish Kumar, Arrvindh Shriraman, and Naveen Vedula. Fusion: Design tradeoffs in coherent cache hierarchies for accelerators. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, 2015.

[97] Monica S. Lam and Robert P. Wilson. Limits of control flow on parallelism. In *ISCA*, 1992.

[98] Maysam Lavasani, Hari Angepat, and Derek Chiou. An fpga-based in-line accelerator for memcached. *IEEE Computer Architecture Letters*, 2013.

[99] Hsien-Hsin S. Lee, Mikhail Smelyanskiy, Chris J. Newburn, and Gary S. Tyson. Stack value file: custom microarchitecture for the stack. In *International Symposium on High Performance Computer Architecture*, 2001.

[100] Yusup Lee, Andrew Waterman, Rimas Avizienis, henry Cook, Chen Sun, Vladimir Stojanov, and Krste Asanovic. A 45nm 1.3ghz 16.7 double-precision gflops/w risc-v processor with vector accelerators. In *ESSCIRC*, 2014.

[101] Jingwen Leng, Tayler H. Hetherington, Ahmed ElTantawy, Syed Zohaib Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. Gpuwattch: enabling energy optimizations in gpgpus. In *ISCA*, 2013.

[102] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO*, 2009.

[103] Kevin T. Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, and Thomas F. Wenisch. Thin servers with smart pipes: designing soc accelerators for memcached. In *ISCA*, 2013.

[104] Hung-Yi Liu and Luca P. Carloni. On learning-based methods for design-space exploration with high-level synthesis. In *DAC*, 2013.

[105] Derek Lockhart, Gary Zibrat, and Christopher Batten. Pymtl: A unified framework for vertically integrated computer architecture research. In *47th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec 2014.

[106] Shan Lu, Pin Zhou, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. Pathexpander: Architectural support for increasing the path coverage of dynamic bug detection. In *MICRO*, 2006.

[107] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *PLDI*, 2005.

[108] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[109] Daniel Lustig and Margaret Martonosi. Reducing gpu offload latency via fine-grained cpu-gpu synchronization. In *High Performance Computer Architecture (HPCA)*, 2013.

[110] Michael J. Lyons, Mark Hempstead, Gu-Yeon Wei, and David Brooks. The accelerator store: A shared memory framework for accelerator-based systems. *TACO*, 2012.

[111] Ikuo Magaki, Moein Khazraee, Luis Vega Gutierrez, and Michael Bedford Taylor. ASIC Clouds: Specializing the Datacenter. In *ISCA*, 2016.

[112] Howard Mao, Sagar Karandikar, Albert Ou, and Soumya Basu. Hardware acceleration of key-value stores. *UC Berkeley CS262a Report*, 2014.

[113] Peter A. Milder, Franz Franchetti, James C. Hoe, and Markus Püschel. Computer generation of hardware for linear digital signal processing transforms. *ACM Transactions on Design Automation of Electronic Systems*, 2012.

[114] Mingxing Tan and Bin Liu and Steve Dai and Zhiru Zhang. Multithreaded Pipeline Synthesis for Data-Parallel Kernels. In *International Conference on Computer Aided Design (ICCAD*, 2014.

[115] Gordon E Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 1965.

[116] Gordon E. Moore. No exponential is forever: but ”forever” can be delayed! In *IEEE International Solid-State Circuits Conference*, 2003.

[117] Thierry Moreau, Mark Wyse, jacob Nelson, Adrian Sampson, Hadi Esmaeilzadeh, Luis Ceze, and Mark Oskin. SNNAP: Approximate computing on programmable socs via neural acceleration. In *HPCA*, 2015.

[118] José M. F. Moura, Jeremy Johnson, Robert W. Johnson, David Padua, Viktor K. Prasanna, Markus Püschel, Bryan Singer, Manuela Veloso, and Jianxin Xiong. Generating platform-adapted DSP libraries using SPIRAL. In *High Performance Embedded Computing (HPEC)*, 2001.

[119] Stephen Neuendorffer and Fernando Martinez-Vallina. Building zynq® accelerators with vivado® high level synthesis. In *FPGA*, 2013.

[120] Rishiyur S. Nikhil. Abstraction in hardware system design. *ACM Queue*, 2011.

[121] Lena E. Olson, Jason Power, Mark D. Hill, and David A. Wood. Border control: Sandboxing accelerators. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, 2015.

[122] Albert Ou, Quan Nguyen, Yunsup Lee, and Krste Asanovic. A case for mvps: Mixed-precision vector processors. In *ISCA Parallelism in Mobile Platforms Workshop*, 2014.

[123] A. Papakonstantinou, K. Gururaj, J.A. Stratton, Deming Chen, J. Cong, and W.-M.W. Hwu. Fcuda: Enabling efficient compilation of cuda kernels onto fpgas. In *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on*, 2009.

[124] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. Architectural support for address translation on gpus. *ACM SIGPLAN Notices*, 49(4):743–758, 2014.

[125] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M Beckmann, Mark D Hill, Steven K Reinhardt, and David A Wood. Heterogeneous system coherence for integrated cpu-gpu systems. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 457–467. ACM, 2013.

[126] Jason Power, Joel Hestness, Marc Orr, Mark Hill, and David Wood. gem5-gpu: A heterogeneous cpu-gpu simulator. *Computer Architecture Letters*, 2014.

[127] Jonathan Power, Mark D Hill, and David A Wood. Supporting x86-64 address translation for 100s of gpu lanes. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 568–578. IEEE, 2014.

[128] A. Putnam, A.M. Caulfield, E.S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G.P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P.Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, 2014.

[129] Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, Christos Kozyrakis, and Mark A. Horowitz. Convolution engine: balancing efficiency & flexibility in specialized computing. In *ISCA*, 2013.

[130] Lawrence Rauchwerger, Pradeep K. Dubey, and Ravi Nair. Measuring limits of parallelism and characterizing its vulnerability to resource constraints. In *MICRO*, 1993.

[131] B. Reagen, R. Adolf, Y.S. Shao, Gu-Yeon Wei, and D. Brooks. MachSuite: Benchmarks for accelerator design and customized architectures. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, 2014.

[132] Brandon Reagen, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. Quantifying Acceleration: Power/Performance Trade-Offs of Application Kernels in Hardware. In *ISLPED*, 2013.

[133] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, Jose Miguel Hernandez-Lobato, Gu-Yeon Wei, and David Brooks. Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators. In *ISCA*, 2016.

[134] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. Dramsim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters*, 2011.

[135] Jack Sampson, Ganesh Venkatesh, Nathan Goulding-Hotta, Saturnino Garcia, Steven Swanson, and Michael Bedford Taylor. Efficient complex operators for irregular codes. In *HPCA*, 2011.

[136] Richard Sampson, Ming Yang, Siyuan Wei, Chaitali Chakrabarti, and Thomas F. Wenisch. Sonic millip3de: A massively parallel 3d-stacked accelerator for 3d ultrasound. In *HPCA*, 2013.

[137] Bob Schaller. The origin, nature, and implications of moore's law. 2014.

[138] Tingting Sha, Milo M. K. Martin, and Amir Roth. Nosq: Store-load communication without a store queue. In *In MICRO*, pages 285–296, 2006.

[139] Ofer Shacham, Megan Wachs, Andrew Danowitz, Sameh Galal, John Brunhaver, Wajahat Qadeer, Sabarish Sankaranarayanan, Artem Vassiliev, Stephen Richardson, and Mark Horowitz. Avoiding game over: Bringing design to the next level. In *Proceedings of the 49th Annual Design Automation Conference*, 2012.

[140] Claude Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 1948.

[141] Yakun Sophia Shao and David Brooks. Energy characterization and instruction-level energy model of Intel's Xeon Phi processor. In *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*, 2013.

[142] Yakun Sophia Shao and David Brooks. ISA-Independent Workload Characterization and its Implications for Specialized Architectures. In *ISPASS*, 2013.

[143] Yakun Sophia Shao and David Brooks. Research Infrastructures for Hardware Accelerators. *Synthesis Lectures on Computer Architecture*, 10(4):1–99, 2015.

[144] Yakun Sophia Shao, Judson Porter, Michael Lyons, Gu-Yeon Wei, and David Brooks. Power, Performance and Portability: System Design Considerations for Micro Air Vehicle Applications. In *Advanced Computer Architecture and Compilation for Embedded Systems (ACACES)*, 2010.

[145] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Aladdin: A Pre-RTL, Power-Performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures. In *ISCA*, 2014.

[146] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. The Aladdin Approach to Accelerator Design and Modeling. In *IEEE Micro*, 2015.

[147] Yakun Sophia Shao, Sam Xi, Viji Srinivasan, Gu-Yeon Wei, and David Brooks. Toward Cache-Friendly Hardware Accelerators. In *Sensors and Cloud Architectures Workshop (HPCA)*, 2015.

[148] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *ASPLOS*, 2002.

[149] Alex Solomatnikov, Amin Firoozshahian, Ofer Shacham, Zain Asgar, Megan Wachs, Wajahat Qadeer, Stephen Richardson, and Mark Horowitz. Using a configurable processor generator for computer architecture prototyping. In *MICRO*, 2009.

[150] Viji Srinivasan, David Brooks, Michael Gschwind, Pradip Bose, Victor Zyuban, Philip N Strenski, and Philip G Emma. Optimizing pipelines for power and performance. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 333–344, 2002.

[151] John A. Stratton, Christopher Rodrigrues, I-Jui Sung, Nady Obeid, Liwen Chang, Geng Liu, and Wen-Mei W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. 2012.

[152] Jeff Stuecheli. Power8 Processor. In *HotChips*, 2013.

[153] Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. Optiml: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 2011.

[154] Arvind K Sujeeth, Kevin J Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Transactions on Embedded Computing Systems (TECS)*, 2014.

[155] Olivier Temam. A defect-tolerant accelerator for emerging high-performance applications. In *ISCA*, 2012.

[156] Kevin B. Theobald, Guang R. Gao, and Laurie J. Hendren. On the limits of program parallelism and its smoothability. In *MICRO*, 1992.

[157] Marc A Unger, Hou-Pu Chou, Todd Thorsen, Axel Scherer, and Stephen R Quake. Monolithic microfabricated valves and pumps by multilayer soft lithography. *Science*, 288(5463):113–116, 2000.

[158] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. Sd-vbs: The san diego vision benchmark suite. IISWC, 2009.

[159] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: reducing the energy of mature computations. *ASPLOS*, 2010.

[160] Ganesh Venkatesh, Jack Sampson, Nathan Goulding-Hotta, Sravanthi Kota Venkata, Michael Bedford Taylor, and Steven Swanson. Qscores: trading dark silicon for scalable energy efficiency with quasi-specific cores. In *MICRO*, 2011.

[161] Huy Vo, Yunsup Lee, Andrew Waterman, and krste Asanovic. A case for os-friendly hardware accelerators. In *ISCA Interaction Between Operating System and Computer Architecture Workshop*, 2013.

[162] David W. Wall. Limits of instruction-level parallelism. In *ASPLOS*, 1991.

[163] Steven J. E. Wilton and Norman P. Jouppi. Cacti: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 1996.

[164] Lisa Wu, Raymond J. Barker, Martha A. Kim, and Kenneth A. Ross. Navigating big data with high-throughput, energy-efficient data partitioning. In *ISCA*, 2013.

[165] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. Q100: The architecture and design of a database processing unit. In *ASPLOS*, 2014.

[166] Sam Likun Xi, Hans Jacobson, Pradip Bose, Gu-Yeon Wei, and David Brooks. Quantifying sources of error in mcpat and potential impacts on architectural studies. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 577–589. IEEE, 2015.

[167] Yakun Sophia Shao and Sam Likun Xi and Vijayalakshmi Srinivasan and Gu-Yeon Wei and David Brooks. Co-Designing Accelerators and SoC Interfaces using gem5-Aladdin. In *MICRO*, 2016.

[168] Praveen Yedlapalli, Nachiappan Chidambaram Nachiappan, Niranjan Soundararajan, Anand Sivasubramaniam, Mahmut T Kandemir, and Chita R Das. Short-circuiting memory traffic in handheld platforms. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 166–177, 2014.

[169] Luke Yen, Stark C. Draper, and Mark D. Hill. Notary: Hardware techniques to enhance signatures. In *International Symposium on Microarchitecture*, 2008.

[170] Takashi Yokota, Kanemitsu Ootsu, and Takanobu Baba. Introducing entropies for representing program behavior and branch predictor performance. In *Workshop on Experimental Computer Science*, 2007.

[171] Kun Yuan, Jae-Seok Yang, and David Z Pan. Double patterning layout decomposition for simultaneous conflict and stitch minimization. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(2):185–196, 2010.

[172] Ning Zhang and Bob Brodersen. The cost of flexibility in systems on a chip design for signal processing applications. *University of California, Berkeley, Tech. Rep*, 2002.

[173] Q. Zhu, B. Akin, H. E. Sumbul, F. Sadi, J. Hoe, L. Pileggi, and F. Franchetti. A 3d-stacked logic-in-memory accelerator for application-specific data intensive computing. In *Proceedings of IEEE International 3D Systems Integration Conference (3DIC)*, 2013.