Approximate Acceleration for a Post Multicore Era

Hadi Esmaeilzadeh

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2013

Reading Committee:

Doug Burger, Chair

Luis Ceze, Chair

Mark Oskin

Program Authorized to Offer Degree:
Department of Computer Science and Engineering

University of Washington

Abstract

Approximate Acceleration for a Post Multicore Era

Hadi Esmaeilzadeh

Co-Chairs of the Supervisory Committee:
Professor Doug Burger
Microsoft Research

Associate Professor Luis Ceze
University of Washington

Starting in 2004, the microprocessor industry has shifted to multicore scaling—increasing the number of cores per die each technology generation—as its principal strategy for continuing performance growth. This work first studies the interplay between the rise of multicore processors and the rise of managed languages—e.g. Java—in the past decade. Then, this dissertation looks into future, studies the trends in transistor scaling, and investigates whether multicore scaling will sustain traditional performance improvements that have been the driving force for the entire computing industry over the past forty years. The results from our work challenges the conventional wisdom that advocates multicore scaling is a viable path for exploiting increased transistor counts and sustaining historical performance trends. Furthermore, the results show that dark silicon—the fraction of chip that needs to be powered off at all times due to power constraints—may break the economics of continued silicon scaling. Our study highlights that radical departures from conventional approaches are necessary to sustain the traditional rate of performance improvements in general-purpose computing. These techniques should provide significant performance and energy efficiency gains across a wide range of applications. This dissertation then proposes a new direction for general-purpose computing that leverages approximation to address the dark silicon challenge. While conventional techniques—such as dynamic voltage and frequency scaling—trade performance for energy, general-purpose approximate computing trades error for both performance and energy gains. We propose variable-precision architectures, a framework from the ISA—Instruction Set Architecture—to the transistor-level implementations that allow conventional von Neumann processors to trade accuracy for energy at the granularity of single instructions. Then, we propose an end-to-end solution, from the programming model to the microarchitecture that leverages an approximate algorithmic transformation to automatically convert a hot code region from a von Neumann model to a neural model. This solution and its associated algorithmic transformation enables a new class of accelerators, called Neural Processing Units (NPUs) with implementation potential in both the digital and the analog domain. This work shows significant gains both in performance and energy when the abstraction of full accuracy is relaxed in general-purpose computing. Furthermore, the proposed approaches open new venues for research in programming languages, architecture, mixed-signal circuit design, and even machine learning. The significant gains from the proposed techniques show that general-purpose approximate computing can be a path forward when the gains from conventional approaches are diminishing.

Table of Contents

List of Figures

List of Tables

Acknowledgments

Looking back, I happily acknowledge that my PhD dissertation is the fruit of years of work under the supervision, guidance, and mentorship of visionary academics. First, I must thank my advisors, Doug Burger and Luis Ceze. Doug tricked me into joining his research group, CART, in 2006. I feel unbelievably lucky to have joined his cult of researchers. His vision, his ambition, and his commitment to hard, risky, and high-impact research have shaped my research personality. His ethics and caring personality have reformed me as a human being. I feel eternally indebted to Doug and admire him as a son admires a caring father. I will always aspire toward becoming a noble human being like Doug. My other advisor, Luis, accepted me in 2010 in his incredible group of students. He patiently worked with me despite me being a high-maintenance guest. He put up with me until UW CSE became my own home. Working with Luis changed my research view and gave me the scientific vision that I was lacking. His uplifting style of advising and inspiring support for creativity are rare gems that will guide me throughout my career as an academic. Luis is the best role model of an incredibly supportive and successful advisor. I hope that I can follow his steps.

Besides my advisors, Kathryn McKinely has played a significant role in my academic career. As she puts it, I am her step-student. Kathryn took me under her support in Austin and guided me toward breaking the barrier of publishing in the top venues. She taught me how to properly communicate research. Thank you Kathryn for advising me even though I was adopted. I have also worked under Karu (Karthikeyan) Sankaralingam's close mentorship. Karu helped me understand how to transform an idea to a high-impact research contribution. His quantitative style of investi-

gating hard problems is my guidelines for conducting scientific research. I should also thank Karu for offering unsolicited advice from time to time to help me become a more mature researcher. I am also greatly thankful to Steve Blackburn for collaborating on a project that seemed will never end. I am also thankful to Steve Keckler for his support when I was in UT Austin.

I have not directly worked with Mark Oskin, but his intellectual presence and critical questioning has strengthened my research foundation. He has always been a balancing force. I am always thankful to his wise, to the point, and timely advices. I also should thank Dan Grossman for showing me how a meticulous researcher analyzes problems and solutions. I also appreciate Scott Hauck's constant eagerness to provide useful and practical advice. I thank Hank Levy, the best department chair who is making UW CSE an engaging, vibrant, and friendly research institute. I also thank Ed Lazowska for his supportive and inspiring role in UW CSE. Lindsay Michimoto makes the life of entire UW CSE family extremely easy. Lindsay was always there to solve any crisis I faced. Knowing that Lindsay is aware of any issue would put my mind at ease. Thank you Lindsay for being the best administrator of all times.

Mark Hill gave me the confidence to follow my life-long dream and become an academic. Now, the dream is becoming a world-renowned scholar like Mark Hill. Thank you Mark for believing in me and voluntarily supporting me. I am deeply grateful to Babak Falsafi who also voluntarily supported my career and helped me to excel. I am also extremely grateful to Lance Fortnow and the rest of Georgia Institute of Technology's faculty who believed in me and hired me.

Meeting Arjang Hassibi was a turning point in my life. He is a true teacher and an incredible life coach. I am lucky to have him in my life. Thank you Arjang for being there when I need a brother to guide me.

I have learned tremendously from my research coconspirators. Adrian Sampson is the best collaborator that one could ever dream of working with. Emily Blem is another dreamy collaborator. It has been a privilege to be working with you two. Thank you for all your contributions. I also thank Ting Cao who is the most hardworking collaborator without whom the project would have never ended. I also should acknowledge the efforts of my other collaborators, Renée St. Amant and Xi Yang.

Joe Devietti was my student mentor when I joined UW CSE. He helped me through a tough transition toward becoming a member of UW CSE family. I am thankful to Joe for showing me how an exemplar graduate student behaves. He is also my role model junior faculty. Thank you Joe for continually helping me even though you have already graduated. Jacob Nelson patiently listened to me whenever I needed to talk. He was there whenever I needed a supportive colleague. Thank you Jacob. Ben Wood is a true example of a welcoming member of UW CSE. Thank you for helping me when I joined. Tom Bergan is the most critical thinking colleague that every student needs. Thanks for always asking hard questions, Tom. Brandon Lucia and I stepped together on the road of finding an academic job. Despite competition, we formed a lasting friendship. Thanks for being such a gracious and noble competitor and colleague.

I am also deeply grateful to Alexander Burger, Aurelia Burger, Lucius Burger, Maximus Burger, Nicki Dell, Carl Ebeling, Ali Farhadi, Mark Gebhart, Boris Grot, Brandon Holt, Maria Jump, Melody Kadenko, Gaya Khachatryan-Grot, Brandon Myers, Gem Naivar, Shahrzad Naraghi, Hal Perkins, Andrew Putnam, Behnam Robatmili, Simha Sethumadhavan, Michelle Silver, Karin Strauss, Olivier Temam. I am also thankful to my friends, Ali Bahram, Ehsan Bahram, Eiman Ebrahimi, Hossein Estiri, Babak Fallahazad, Bahman Hekmatshoar Tabari, Hossein Hooshyar, Amir Hosseini, Ali Jalali, Sied Raed Mousavi, Ardavan Pedram, Sajjad Pouraryan, Davood Shahrjerdi, Seyed Reza Yousefi.

I owe everything to my family. I am happy that I am keeping the family tradition and following my father's footsteps by becoming an academic. My father, Hamdollah Esmaeilzadeh, my mom, Fatemeh Basiri, my brothers, Kazem, Mehdi, and Mohammad Amin Esmaeilzadeh, my in-laws, Ahamd Hosseini Porgahm, Najibeh Moghadam, Sepideh Hosseini Porgham, and Mohammad Shahab Hosseini Porgham.

Last but not least, the love of my life, Sama (Somayeh) Hosseini Porgahm, without whom I could not have survived PhD. The words cannot express my gratitude toward you.

Dedication

to the blue sky of my life, Sama

CHAPTER 1

INDUSTRY OF NEW POSSIBILITIES

Computing has become a pervasive commodity. However, there is a basic difference between the computing industry and other commodity industries: consumers buy computing products because they *improve* while consumers buy commodities because they run out of them. It is not just the devices that are improving; it is the offered services and experiences that consistently improves. As Figure 1.1 conceptually illustrates, the computing industry is not an industry of replacement; it is an industry of new possibilities. One of the primary drivers of this economic model is the exponential reduction in the cost of performing general-purpose computing. While in 1971, at the dawn of microprocessors, the price of 1 MIPS[1] was roughly $5000, it today is about 4¢. This is an exponential reduction in the cost of raw material for computing. This continuous and exponential reduction in cost has formed the basis of computing industry's economy in the past four decades.

The primary enabling factor of this economic model is the consistent and exponential improvement in transistor fabrication technology that happens every 18 month—Moore's Law [84]. Computer architects have made these device-level improvements available to the rest of the computing stack by building and continuously speeding up general-purpose processors. The exponential improvements in the performance of general-purpose processors have resulted in the exponential reduction in the cost of computing and led our industry down the path of becoming an industry of new possibilities. However, the traditional rates of improvements in transistor fabrication broke in mid 2000's. This break down in turn broke many techniques that used to improve the perfor-

[1] MIPS: Million Instruction Per Second

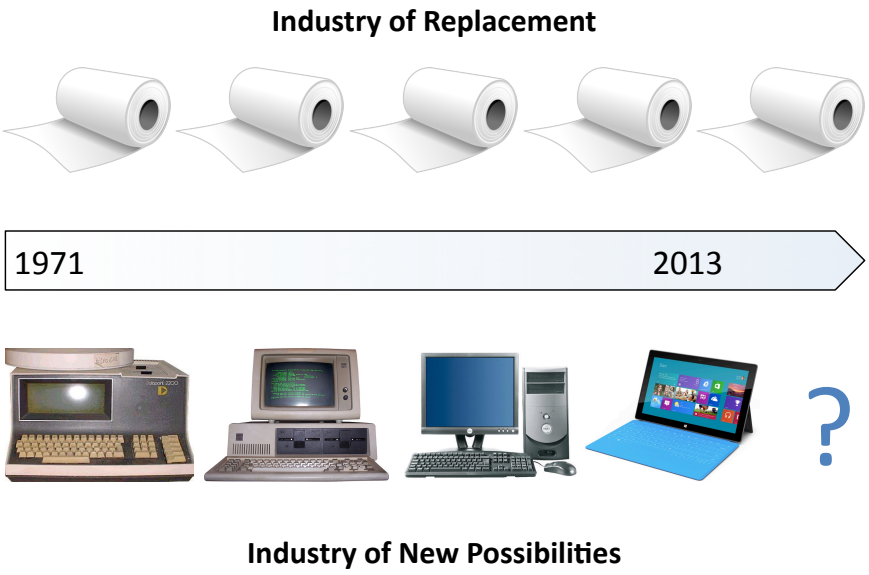


Figure 1.1: The industry of new possibilities versus an industry of replacement.

mance of single-core general-purpose processors. This issue caused the microprocessor division of the computing industry to race down the path of consistently integrating more cores on the general-purpose processors. Multicore processors became the primary driver of performance improvement. The general consensus is formed around the idea that by exploiting parallelism in the applications, we can overcome the trends in the transistor level. Many in the community believed that the computing industry has entered a long multicore era. Multicore scaling—integrating more processor cores every technology generation—is believed to provide benefits that can sustain the economic model of our industry. This dissertation questions this consensus and aims to provide answers to the following timely and fundamental question that are posed to the entire computer science and engineering community.

Can our industry rely on multicore scaling and still continue being an industry of new possibilities?
Is the computing industry on the brink of becoming an industry of replacement?
What is a possible path forward in general-purpose computing that may avert becoming an industry of replacement?

This dissertation answers the above questions by following these four steps:

1. It briefly discusses Moore's Law [84] and illustrates how the advances in transistor fabrication and general-purpose processor design has led our industry to become an industry of new possibilities (Chapter 1).

2. It looks back to the past decade and studies how the current paradigm of general-purpose processor design, multicore processors, has interplayed with modern workloads and programming languages (Chapter 2).

3. It studies the trends in the transistor level and shows how the computing industry may be on the brink of becoming an industry of replacement if we continue building the general-purpose processors the same way we do today (Chapter 3).

4. It finally proposes a new way of doing computation, general-purpose approximate computing, that could be one possible path that averts becoming an industry of replacement (Chapter 4, Chapter 5).

1.1 Moore's Law Enables New Possibilities

Moore's Law [84]—doubling the number of transistors every 18 months—has been a fundamental driver of computing for more than four decades. Over the past 40 years, every 18 month, the transistor manufacturing facilities have been able to develop a new technology generation that doubles the number of transistors on a single monolithic chip. However, doubling the number of transistors does not provide any benefits by itself. Computer architects harvest these transistors and design *general-purpose processors* that make these tiny switches available to the rest of computing community. By building general-purpose processors, the computer architecture community provides the link, the mechanisms, and the abstractions that make these devices accessible to compilers, programming languages, system designers, and application developers. To this end, general-purpose processing has enabled the computing industry to commodify computing and make it pervasively present everywhere. Computer architecture has also been able to harvest the exponentially increasing number of transistor and traditionally almost with the same rate improve the performance of general-purpose processors. This consistent improvements in performance has proportionally

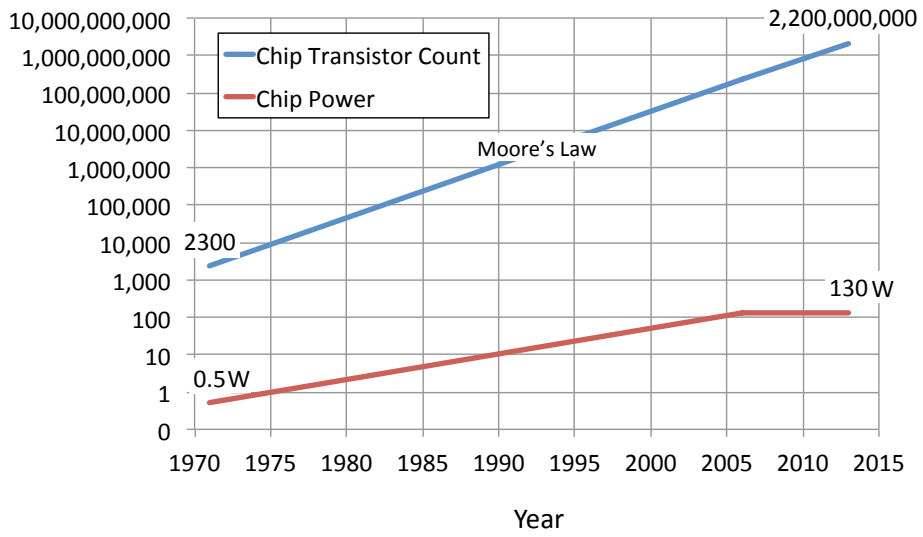


Figure 1.2: Moore's Law and chip power since the dawn of microprocessors. Even though, the number of transistors on a single chip has exponentially increased the required power to turn them on has not followed the same rate and in fact, has plateaued in the past few years.

reduced the cost of computing that in turn enabled the application and system developers to consistently offer new possibilities. The ability to consistently provide new possibilities has paid off the huge cost of developing new process technologies for transistor fabrication. This sustaining loop has sustained the economic model of our industry over the course of the past four decades. Even though this economic loop seems relatively straightforward, there are fundamental challenges that are associated with integrating exponentially increasing number of transistors on a single chip.

1.2 Dennard Scaling Enables Moore's Law

One of the main challenges of doubling the number of transistors on the chip is powering them on without melting the chip. As Figure 1.2 illustrates even though the number of transistors on the chip has exponentially increased since 1971—the time first microprocessors were introduced—the chip power has merely increased very modestly and has plateaued in the recent years. It seems almost magical that the same amount of power can switch double the number of transistors. Robert Dennard formulated how the new transistor fabrication process technology can provide such physical properties [26]. In fact, Dennard's theory of scaling is the main force behind Moore's Law.

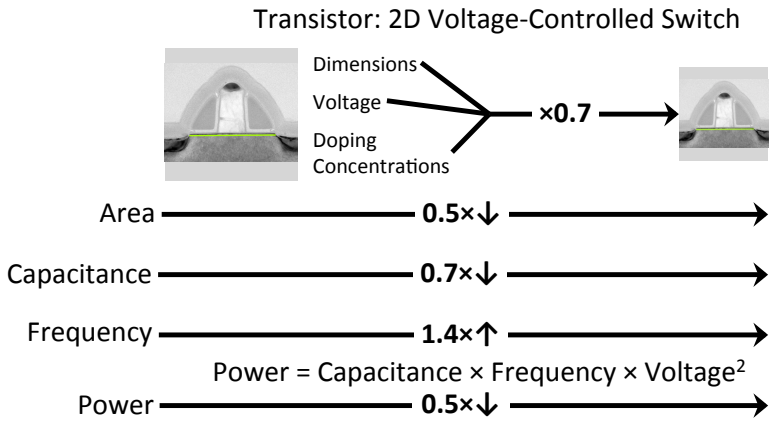


Figure 1.3: Dennard's constant-field scaling.

Based on Dennard's scaling theory, a transistor is a 2-dimensional voltage-controlled switch. As Figure 1.3 depicts, according to Dennards' theory, consistently scaling the dimensions, the operating voltages, and the doping concentrations of a transistor by a factor $\alpha = \sqrt{2}$, will result in a transistor that has $\frac{1}{\alpha^2} = \frac{1}{2}$ the area of the original transistor; its capacitance will decrease by a factor of $\alpha = \sqrt{2}$; and it can be toggled with $\frac{1}{\alpha} = \frac{1}{\sqrt{2}} = 1.4$ times higher frequency. However, the electric field in the transistor remains constant. Dennard's constant-field scaling roughly formulates how the transistors form a new process technology will behave.

The dynamic power of transistors follows $Capacitance \times Frequency \times Voltage^2$. Thus, scaling will decrease the switching power of transistors by a factor of $\frac{1}{\alpha^2} = \frac{1}{2}$. That is, the power of transistors will decrease by the same rate their area shrinks. Therefore, a new process technology can double the number of transistors in a fixed chip area without increasing the power consumption of chip, i.e. Moore's Law. However, in mid 2000's Dennard scaling failed. The transistor operating voltages could not be scaled with the traditional rates, i.e. $\frac{1}{2}$, due to leakage current. Furthermore, due to physical limitations, the capacitance of transistors was not decreasing with the historical rates. As a result, the power of the transistors would not decrease with the same rate the area of the transistors was shrinking. The only option to avoid increases in the chip power consumption was to not increase the clock frequency or even lower it.

Before, we lay these scaling trends of transistors on the evolution history of microprocessors, we introduce a phenomenon called *dark silicon* [28]. When developing a new process technology, if the rate that the power of the transistors scales is less than the rate the area of the transistors

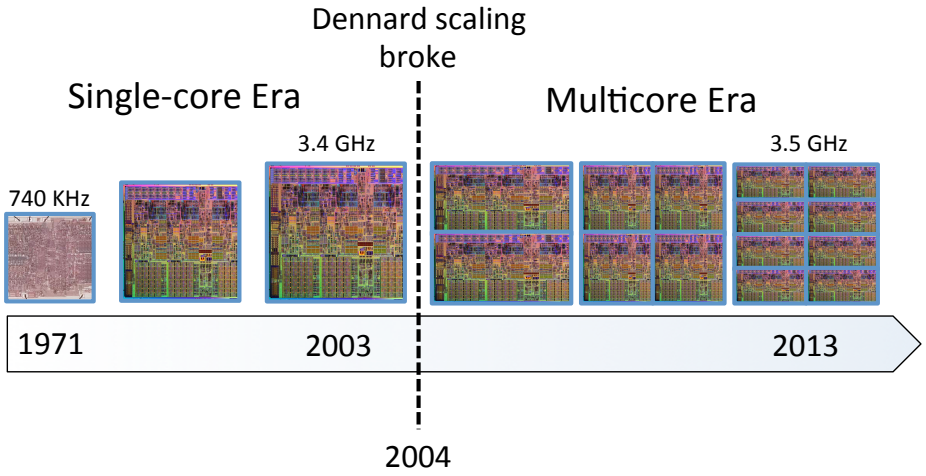


Figure 1.4: End of Dennard scaling marks the beginning of multicore era. However, the crucial and timely question is: are multicore a long-term solution or just a stop-gap?

shrinks, it might not be possible to turn on and utilize all the transistors that scaling provides. Thus, we define dark silicon as follows.

Dark silicon is the fraction of chip that needs to be powered off at all times due to the power constraints.

As we will discuss in Chapter 3, the low utilities of this dark silicon poses a great challenge for the entire computing community. If we cannot utilize the transistors that developing costly new process technologies provide, how can we justify their development cost? If we cannot utilize the transistors to improve the performance of the general-purpose processors and reduce the cost of computing, how can we avert becoming an industry of replacement? This dissertation quantifies the severity of the dark silicon problem and proposes a new possible direction to tackle it.

1.3 End of Dennard Scaling and Multicore Era

For a long time, when the computing industry was under the reign of Dennard scaling, computer architects harvested the new transistors to not only build higher frequency single-core microprocessors, but also equip them with more capabilities. For example, as technology scaled, the processors were packing better branch predictors, wider pipelines, larger caches, and etc. However, Dennard scaling broke in mid 2000's at 90 nm [27] that created a power density problem. The power density

problem in turn broke many of the techniques that were used to improve the performance of single-core processors. The industry raced down the path of building multicore processors. The multicore era started at 2004, when the major consumer processor vendor (Intel) cancelled its next generation single-core microarchitecture, Prescott, and gave up on focusing exclusively on single-thread performance switching to multicore, as their performance scaling strategy.

We mark the start of multicore era not with the date of the first multicore part, but with the time multicore processors became the default and main strategy.

The basic idea behind designing multicore processors was to substitute building more complex/-capable single-cores processors with building multicore processors that constitute simpler and/or lower frequency cores. Many in the community believe that by exploiting parallelism in the applications, we can overcome the trends in the transistor level. The general consensus is that a long-term era of multicore has begun and by increasing the number of cores every technology generation, processors will provide benefits that will enable developing many more process fabrication technologies. Many believe that there will be thousands of cores on each single chip. However, until our dark silicon ISCA paper [32], there is quantitative study that showed how sever the problem is at the transistor level and how the transistor scaling trends will affect the prospective benefits from multicore processors. In Chapter 3, we quantitatively question the consensus about multicore scaling. The results show that even with optimistic assumptions, multicore processors scaling in not a long-term solution and cannot sustain the historical rates of performance growth in the coming years. The gap between the projected performance of multicore processors and what the microprocessor industry has historically provided is significantly large, 24$\times$. Due to lack of high degree of parallelism and the severe degree of energy efficiencies in the transistor level, adding more cores will not even enable using all the transistors that new process technologies provide. In only half a decade from now, more than 50% of the chip will be dark. The lack of performance benefits and the lack of ability to utilize all the transistors that new process technologies provide may undermine the economic viability of developing new technologies. We may stop scaling not because of the physical limitations, but because of the economics. Moore's Law that has worked as a clock and enabled the computing industry to consistently provide new possibilities and periodically may stop or slow down significantly. The entire computing industry may be at the brink of becoming an

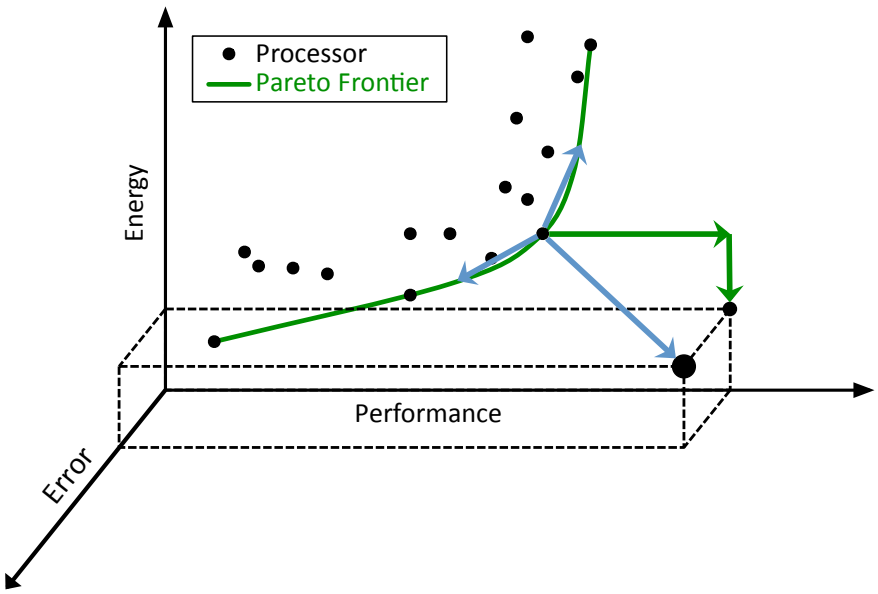


Figure 1.5: While conventional approaches trade energy for performance, general-purpose approximate computing explores the error dimension. By allowing error to happen in the computation, approximate architectures can potentially improve both performance and energy efficiency.

industry of replacement if new avenues for computing are not discovered. A shift of focus seems essential in the computer architecture community.

Thesis statement: *The results from this work shows that the current paradigm of general-purpose processor design, multicore processors, may not sustain the traditional trend of performance scaling. The gap is so wide that radical departures from conventional approaches are necessary to provide performance and efficiency gains across many classes of applications. The objective is to develop techniques that can squeeze and extract as much energy efficiency and performance as possible from the silicon that can be turned on. This dissertation proposes general-purpose approximate computing as one possible path forward for this post multicore era.*

1.4 General-Purpose Approximate Computing for a Post Multicore Era

Relaxing the high tax of providing perfect accuracy at the device, circuit, architecture, and programming language levels can provide significant opportunities to improve performance and energy efficiency for the domains in which applications can tolerate approximation [80, 25, 83, 101, 15, 4]. There is in fact an emerging opportunity for computer architecture design due to the synergy between applications that can tolerate inaccurate computation and the unreliability in the computation fabric at the transistor level. These applications span embedded systems that operate on sensory inputs to multimedia, vision, web search, machine learning, optimization, big data analytics, and many more. As Figure 1.5 conceptually illustrates, while conventional techniques—such as dynamic voltage and frequency scaling—trade performance for energy, general-purpose approximate computing trades error for gains in performance *and* energy. Four broad categories of applications can benefit from general-purpose approximate computing:

1. Applications with analog inputs such as sensory data processing and scene reconstructions in augmented reality

2. Applications with analog output such as multimedia

3. Applications with multiple acceptable answers such as web search and machine learning

4. Convergent applications such as data analytics and optimization.

These diverse classes of applications provide opportunities for general-purpose approximate computing on both mobile and cloud computing systems. As Figure 1.6 depicts, adding the dimension of error in the design space of processors converts the problem of finding the Pareto frontier to finding the Pareto surface. Navigating this three-dimensional space and finding designs that form this Pareto surface is a fascinating research direction. This dissertation explores this three dimensional space and proposes two novel techniques that cover parts of the Pareto surface.

Applying approximation without discipline would make the construction of reliable software nearly impossible and could lead to catastrophic failures during execution. For approximate computation to be safe, it must be confined to the error-tolerant parts of the program. It must not,

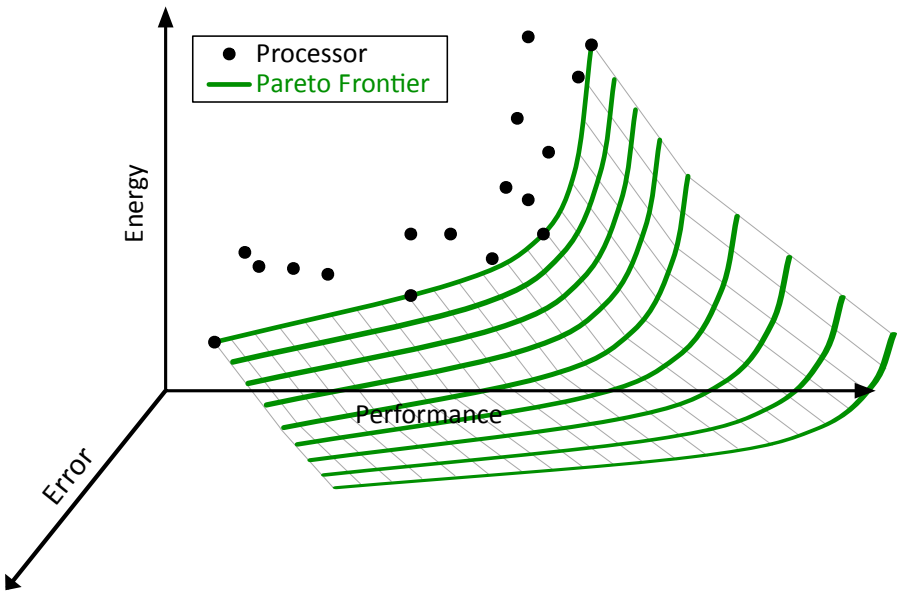


Figure 1.6: Adding the dimension of error turns the problem of finding the Pareto frontier to finding the Pareto surface. Navigating this three-dimensional space and finding the shape of this Pareto surface is a fascinating research direction.

for example, lead to uncontrolled jumps or wild pointers. This need for a balance between approximate and traditional execution has led to research on disciplined approaches to approximation. In the programming language level, Sampson et al. introduced the EnerJ approximation-aware language that allows programmers safely distinguish error-tolerant program components and to protect critical components from errors [98]. At the architecture level, we introduce a variable-precision Instruction Set Architecture (ISA) that allows conventional von Neumann processors to interleave approximate and precise operations at the granularity of single instructions (Chapter 4). This ISA allows the compiler to convey what can be approximated without specifying how, permitting the microarchitecture to choose from a range of approximation techniques without exposing them to software. We also designed the dual-voltage Truffle microarchitecture that implements this variable-precision ISA.

While simulation results for that architecture showed energy savings up to 43%, the efficiency gains for this approach are limited by the constraints of traditional processor design. Only part of the von Neumann pipeline can be optimized with this approach—the front end, including instruction decode and control, cannot be approximated. Contrastingly, Chapter 5 introduces an algorith-

mic transformation that converts an approximable code region from a von Neumann model to a neural model, enabling much larger performance and efficiency gains. Leveraging this algorithmic transformation, we introduce a new class of accelerators, called neural processing units (NPUs). NPUs achieve an average 2.3× speedup and 3.0× energy savings for general-purpose approximate programs. This new class of accelerators shows that significant performance and efficiency gains are possible when the abstraction of full accuracy is relaxed in general-purpose computing.

1.5 Contributions

This dissertation makes the following contributions:

1. **Multicores, measured power, and modern workloads.** Through rigorous power and performance measurements for a diverse set of benchmarks on eight different processors, this work shows that commonly used benchmarks do not predict the power, performance, energy trends in modern applications, which are mostly implemented in managed programing languages. This work embodies a rigorous and systematic exploration of power, performance, and workload across technology generations using measured power. It quantifies the extent of some known and many unobserved hardware and software trends, pointing in new research directions.

2. **Dark silicon and the end of multicore era.** This dissertation challenges the conventional wisdom that suggests multicore scaling—increasing the number of cores every new technology generation—is the viable path for exploiting increased transistor counts and sustaining the historical performance trends. We show that multicore chips will be increasingly power-limited with each new technology generation and an increasing fraction of chips have to be powered off at all times. This phenomenon is known as dark silicon. This work also shows that multicore scaling provides much less performance gain than conventional wisdom suggests. These results show that without a breakthrough in process technology or microarchitecture, directions beyond multicore are necessary to continue the historical rate of performance improvement and justify the economic viability of continued silicon scaling.

3. **Variable-precision von Neumann architectures.** To address these challenges, we designed a novel framework for conventional general-purpose processors to trade accuracy for energy at the granularity of single instructions. We introduced a variable-precision ISA as an extension to a conventional ISA that allows a conventional microarchitecture to run many interleavings of approximate and precise instructions. This ISA allows the compiler to convey what can be approximated without specifying how. This abstraction allows the microarchitecture to freely choose from a range of approximation techniques without exposing them to the compiler or programmer. We also designed a dual-voltage microarchitecture that implements the proposed ISA and trades accuracy for energy on statically marked operations and storage.

4. **From a von Neumann model to a hybrid von Neumann-neural model of computing.** We proposed a new acceleration technique that leverages a simple programmer annotation ("approximable") to transform a hot code region from a von Neumann model to a neural model. To this end, we proposed an algorithmic transformation that automatically selects and trains a neural network to mimic a region of imperative code. After the learning phase, the compiler transparently replaces the original code with an invocation of a low-power accelerator. Leveraging this transformation, we introduced a new class of approximate accelerators, called Neural Processing Units (NPUs), with implementation potential in both the digital and the analog domain. An NPU is a reconfigurable custom hardware that is tightly coupled to the processor pipeline to efficiently execute the neural networks resulted from the algorithmic transformation. This works shows significant gains both in performance and energy when the abstraction of full precision is relaxed in general-purpose computing.

The results from our dark silicon study show that the current paradigm of general-purpose processor design cannot deliver the traditional performance benefits to sustain the economics of computing industry. Additionally, the significant gains from the proposed techniques show that general-purpose approximate computing is one possible path forward when the benefits from conventional approaches are diminishing. This dissertation shows that there is a tremendous opportunity for innovation in general-purpose approximate computing that can yield in gains that are much needed by the entire computer science and engineering community.

CHAPTER 2

LOOKING BACK: MULTICORES, MEASURED POWER, AND MODERN WORKLOADS

The past decade has delivered two significant shifts. (1) Microprocessor design has been transformed by the limits of chip power and Dennard scaling—leading to multicore processors. (2) Managed languages and an entirely new software landscape emerged—revolutionizing how software is deployed, is sold, and interacts with hardware. Researchers most often examine these changes in isolation. Architects mostly grapple with microarchitecture design through the narrow software context of native sequential SPEC CPU benchmarks, while language researchers mostly consider microarchitecture in terms of performance alone. This chapter explores the clash of these two shifts over the past decade by measuring power, performance, energy, and scaling, and considers what the results may mean for the future. Our diverse findings include: (a) native sequential workloads do not approximate managed workloads or even native parallel workloads; (b) diverse application power profiles suggest that future applications and system software will need to participate in power optimization and management; and (c) software and hardware researchers need access to real measurements to optimize for power and energy.
This chapter is based on work presented in ASPLOS (2011) [33], IEEE Micro Top Picks (2012) [37], and Communications of ACM Research Highlights (2012) [38]. The work is a result of collaboration with Ting Cao[a], Xi Yang[a], Steve Blackburn[a], and Kathryn McKinely[b].

[a] Australian National University

[b] The University of Texas at Austin, Microsoft Research

2.1 Introduction

Quantitative performance analysis is the foundation for computer system design and innovation. In their classic paper, Emer and Clark noted that "A lack of detailed timing information impairs efforts to improve performance" [29]. They pioneered the quantitative approach by characterizing instruction mix and cycles per instruction on timesharing workloads. They surprised expert reviewers by

demonstrating a gap between the theoretical 1 MIPS peak of the VAX-11/780 and the 0.5 MIPS it delivered on real workloads. Industry and academic researchers in software and hardware all use and extend this principled performance analysis methodology. This chapter applies this quantitative approach to measured power. This work is timely because the past decade heralded the era of power and energy constrained hardware design.[1] Furthermore, demand for energy efficiency has intensified in large-scale systems, in which energy began to dominate costs, and in mobile systems, which are limited by battery life. A lack of detailed energy measurements is impairing efforts to reduce energy consumption on modern workloads.

Society has benefited enormously from exponential hardware performance improvements. Moore observed that transistors will be smaller and more numerous in each new generation [84]. For a long time, this simple rule of integrated circuit fabrication came with an exponential and transparent performance dividend. Shrinking a transistor lowers its gate delay, which raises the processor's theoretical clock speed (Dennard scaling [10]). Until recently, shrinking transisters delivered corresponding clock speed increases and more transistors in the same chip area. Architects used the transistor bounty to add memory, prefetching, branch prediction, multiple instruction issue, and deeper pipelines. The result was *exponential* single-threaded performance improvements.

Unfortunately physical power and wire-delay limits derailed the clock speed bounty of Moore's law in current and future technologies. Power is now a first-order hardware design constraint in all market segments [85]. Power constraints have severely limited clock scaling and projections show that it will prevent utilizing all the transistors simultaneously [32, 57]. In addition, the physical limitations of wires prevent single-cycle access to a growing number of the transistors on a chip [64]. To effectively use more transistors at smaller technologies, these limits forced manufactures to turn to multicore processors and recently to heterogeneous parallel systems—e.g., integrated GPUs and CPUs—that seek power efficiency through specialization. Parallel heterogeneous hardware requires parallel software and exposes software developers to ongoing hardware upheaval. Unfortunately, most software today is not parallel, nor is it designed to modularly decompose onto such heterogeneous substrate.

---

[1] Energy $=$ power $\times$ execution time.

Moore's transistor bounty also drove *orthogonal* and *disruptive* changes in how software is deployed, is sold, and interacts with hardware over this same decade. Demands for correctness, complexity management, programmer productivity, time-to-market, reliability, security, and portability pushed developers away from low-level compiled ahead-of-time (*native*) programming languages. Developers increasingly choose high-level *managed* programming languages with a selection of safe pointer disciplines, garbage collection (automatic memory management), extensive standard libraries, and dynamic just-in-time compilation for hardware portability. For example, modern web services combine managed languages, such as PHP on the server side and JavaScript on the client side. In markets as diverse as financial software and cell phone applications, Java and .NET are the dominant choices. The exponential performance improvements provided by hardware hid many of the costs of high-level languages and helped create a virtuous cycle with ever more capable and high-level software. This ecosystem is resulting in an explosion of developers, software, and devices that continue to change how we live and learn.

Unfortunately, a lack of power measurements is impairing efforts to reduce energy consumption on traditional and modern software.

2.2 Overview

This chapter quantitatively examines power, performance, and scaling during this period of disruptive software and hardware changes (2003–2011). Voluminous research explores performance analysis and a growing body of work explores power (see Chapter 6), but our work is the first to systematically measure the power, performance, and energy characteristics of software and hardware across a range of processors, technologies, and workloads.

We execute 61 diverse sequential and parallel benchmarks written in three native languages and one managed language, all widely used: C, C++, Fortran, and Java. We choose Java because it has mature virtual machine technology and substantial open source benchmarks. We choose eight representative Intel IA32 processors from five technology generations (130 nm to 32 nm). Each processor has an isolated processor power supply with stable voltage on the motherboard, to which we attach a Hall effect sensor that measures power supply current, and hence processor power. We calibrate and validate our sensor data. We find that power consumption varies widely

among benchmarks. Furthermore, relative performance, power, and energy are not well predicted by core count, clock speed, or reported Thermal Design Power (TDP). TDP is the nominal amount of power the chip is designed to dissipate (i.e., without exceeding the maximum transistor junction temperature).

Using controlled hardware configurations, we explore the energy impact of hardware features and workload. We perform historical and Pareto analyses that identify the most power- and performance-efficient designs in our architecture configuration space. We make all of our data publicly available in the ACM Digital Library [34] as a companion to our original ASPLOS 2011 paper. Our data quantifies a large number workload and hardware trends with precision and depth, some known and many previously unreported. This chapter highlights eight findings, which we list in Figure 2.1. Two themes emerge from our analysis: *workload* and *architecture*.

Workload. The power, performance, and energy trends of native workloads substantially differ from managed and parallel native workloads. For example, (a) the SPEC CPU2006 native benchmarks draw significantly less power than parallel benchmarks; and (b) managed runtimes exploit parallelism even when executing single-threaded applications. The results recommend that systems researchers include managed and native, sequential and parallel workloads when designing and evaluating energy-efficient systems.

Architecture. Hardware features such as clock scaling, gross microarchitecture, simultaneous multithreading, and chip multiprocessors each elicit a huge variety of power, performance, and energy responses. This variety and the difficulty of obtaining power measurements recommends exposing on-chip power meters and, when possible, power meters for individual structures, such as cores and caches. Modern processors include power management techniques that monitor power sensors to minimize power usage and boost performance. However, only in 2011 (after our original paper) did Intel first expose energy counters, in their production Sandy Bridge processors. Just as hardware event counters provide a quantitative grounding for performance innovations, future architectures should include power and/or energy meters to drive innovation in the power-constrained computer systems era.

Findings

Power consumption is highly application dependent and is poorly correlated to TDP.

Power per transistor is relatively consistent within microarchitecture family, independent of process technology.

Energy-efficient architecture design is very sensitive to workload. Configurations in the native non-scalable Pareto Frontier substantially differ from all the other workloads.

Comparing one core to two, enabling a core is not consistently energy efficient.

The Java Virtual Machine induces parallelism into the execution of single-threaded Java benchmarks.

Simultaneous multithreading delivers substantial energy savings for recent hardware and for in-order processors.

The most recent processor in our study does not consistently increase energy consumption as its clock increases.

Controlling for technology, hardware parallelism, and clock speed, the out-of-order architectures have similar energy efficiency as the in-order ones.

Figure 2.1: This chapter focuses on eight findings from an analysis of measured chip power, performance, and energy on 61 workloads and eight processors. The ASPLOS paper includes more findings and analysis.

2.3 Methodology

This section overviews essential elements of our methodology. We refer the reader to the ASPLOS (2011) paper [33] for a more detailed treatment.

Software. We systematically explore workload selection and show that it is a critical component for analyzing power and performance. Native and managed applications embody different trade-offs between performance, reliability, portability, and deployment. It is impossible to meaningfully separate language from workload and we offer no commentary on the virtue of language choice. We create four *workloads* from 61 benchmarks.

Native non-scalable. C, C++, and Fortran single-threaded compute-intensive benchmarks from SPEC CPU2006.

Native scalable. Multithreaded C and C++ benchmarks from PARSEC.

Java non-scalable. Single and multithreaded benchmarks that do not scale well from SPECjvm, DaCapo 06-10-MR2, DaCapo 9.12, and pjbb2005.

Java scalable. Multithreaded Java benchmarks from DaCapo 9.12 that scale in performance similarly to native scalable on the i7 (45).

We execute the Java benchmarks on the Oracle HotSpot 1.6.0 virtual machine because it is a mature high performance virtual machine. The virtual machine dynamically optimizes each benchmark on each architecture. We use best practices for virtual machine measurement of steady state performance [9]. We compile the native non-scalable workload with *icc* at `-o3`. We use *gcc* at `-o3` for the native scalable workload because *icc* did not correctly compile all benchmarks. The *icc* compiler generates better performing code than *gcc*. We execute the same native binaries on all machines. All the parallel native benchmarks scale up to eight hardware contexts. The Java scalable workload is the subset of Java benchmarks that scale well.

Hardware. Table 2.1 lists our eight Intel IA32 processors which cover four process technologies (130 nm, 65 nm, 45 nm, and 32 nm) and four microarchitectures (NetBurst, Core, Bonnell, and Nehalem). The release price and date give context regarding Intel's market placement. The Atoms and the Core 2Q (65) *Kentsfield* are extreme market points. These processors are only examples of many processors in each family. For example, Intel sells over 60 Nehalems at 45 nm, ranging in price from around $190 to over $3700. We believe these samples are representative because they were sold at similar price points.

To explore the influence of architectural features, we selectively down-clock the processors, disable cores on these multicore processors, disable simultaneous multithreading (SMT), and disable Turbo Boost using BIOS configuration.

Power, performance, and energy measurement. We measure on-chip power by isolating the direct current (DC) power supply to the processor on the motherboard. Prior work used a clamp ammeter, which can only measure the whole system alternating current (AC) supply [73, 66]. We use *Pololu's ACS714* current sensor board. The board is a carrier for *Allegro's* $\pm 5A$ *ACS714* Hall effect-based linear current sensor. The sensor accepts a bidirectional current input with a magnitude up to $5A$. The output is an analog voltage ($185mV/A$) centered at $2.5V$ with a typical error of less than 1.5%. We place the sensors on the $12V$ power line that supplies only the processor. We

measure voltage and find that it is very stable, varying less than 1%. We send the values from the current sensor to the machine's USB port using a *Sparkfun's Atmel AVR Stick*, a simple data-logging device with a data-sampling rate of 50 Hz. We used a similar arrangement with a 30 A Hall effect sensor for the high power i7 (45). We execute each benchmark, log its power values, and then compute average power consumption. After publishing the original paper, Intel made chip-level and core-level energy measurements available on Sandy Bridge processors [23]. Our methodology should slightly overstate chip power because it includes losses due to the motherboard's voltage regulator. Validating against the Sandy Bridge energy counter shows that our power measurements consistently measure about 5% more current.

We execute each benchmark multiple times on every architecture, log its power values, and then compute average power consumption. The aggregate 95% confidence intervals of execution time and power range from 0.7 to 4%. The measurement error in time and power for all processors and benchmarks is low. We compute arithmetic means over the four workloads, weighting each workload equally. To avoid biasing performance measurements to any one architecture, we compute a *reference performance* for each benchmark by averaging the execution time on four architectures: Pentium 4 (130), Core 2D (65), Atom (45), and i5 (32). These choices capture four microarchitectures and four technology generations. We also normalize energy to a reference, since energy = power $\times$ time. The reference energy is the average benchmark power on the four processors multiplied by their average execution time.

We measure the 45 processor configurations (8 stock and 37 BIOS configurations) and produce power and performance data for each benchmark and processor. Figure 2.2 shows an example of this data, plotting the power versus performance characteristics for one of the 45 processor configurations, the stock i7 (45).

Table 2.1: Specifications for the eight processors used in the experiments.

Processor	μArch	Processor	sSpec	Release Date	Price USD	CMP SMT	LLC B	Clock GHz	nm	Trans M	Die mm^2	VID Range V	TDP (W)	FSB MHz	B/W GB/s	DRAM Model
Pentium 4	NetBurst	Northwood	SL6WF	May '03	—	1C2T	512K	2.4	130	55	131	—	66	800	—	DDR-400
Core 2 Duo E6600	Core	Conroe	SL9S8	Jul '06	$316	2C1T	4M	2.4	65	291	143	0.85 - 1.50	65	1066	—	DDR2-800
Core 2 Quad Q6600	Core	Kentsfield	SL9UM	Jan '07	$851	4C1T	8M	2.4	65	582	286	0.85 - 1.50	105	1066	—	DDR2-800
Core i7 920	Nehalem	Bloomfield	SLBCH	Nov '08	$284	4C2T	8M	2.7	45	731	263	0.80 - 1.38	130	—	25.6	DDR3-1066
Atom 230	Bonnell	Diamondville	SLB6Z	Jun '08	$29	1C2T	512K	1.7	45	47	26	0.90 - 1.16	4	533	—	DDR2-800
Core 2 Duo E7600	Core	Wolfdale	SLGTD	May '09	$133	2C1T	3M	3.1	45	228	82	0.85 - 1.36	65	1066	—	DDR2-800
Atom D510	Bonnell	Pineview	SLBLA	Dec '09	$63	2C2T	1M	1.7	45	176	87	0.80 - 1.17	13	665	—	DDR2-800
Core i5 670	Nehalem	Clarkdale	SLBLT	Jan '10	$284	2C2T	4M	3.4	32	382	81	0.65 - 1.40	73	—	21.0	DDR3-1333

2.4 Perspective

We organize our analysis into seven findings, as summarized in Figure 2.1. The ASPLOS (2011) paper [33] contains additional analyses and findings. We begin with broad trends. We show that applications exhibit a large range of power and performance characteristics that are not well summarized by a single number. This section conducts a Pareto energy efficiency analysis for all of the 45 nm processor configurations. Even with this modest exploration of architectural features, the results indicate that each workload prefers a different hardware configuration for energy efficiency.

2.4.1 Power is Application Dependent

The nominal thermal design power (TDP) for a processor is the amount of power the chip may dissipate without exceeding the maximum transistor junction temperature. Table 2.1 lists TDP for each processor. Because measuring real processor power is difficult and TDP is readily available, TDP is often substituted for real measured power. Figure 2.3 shows that this substitution is problematic. It plots on a logarithmic scale, measured power for each benchmark on each stock processor as a function of TDP and indicates TDP with an '✗'. TDP is strictly higher than actual power. The gap between peak measured power and TDP varies from processor to processor and TDP is up to a factor of four higher than measured power. The variation among benchmarks is highest on the i7 (45) and i5 (32), likely reflecting their advanced power management. For example on the i7 (45), measured power varies between 23W for 471.omnetpp and 89W for fluidanimate! The smallest variation between maximum and minimum is on the Atom (45) at 30%. This trend is not new. All the processors exhibit a range of benchmark-specific power variation. TDP loosely correlates with power consumption, but it *does not* provide a good estimate for (1) maximum power consumption of individual processors, (2) comparing among processors, or (3) approximating benchmark-specific power consumption.

Power consumption is highly application dependent and is poorly correlated to TDP.

Figure 2.2 plots power versus relative performance for each benchmark on the i7 (45), which has eight hardware contexts and is the most recent of the 45 nm processors. Native (red) and managed

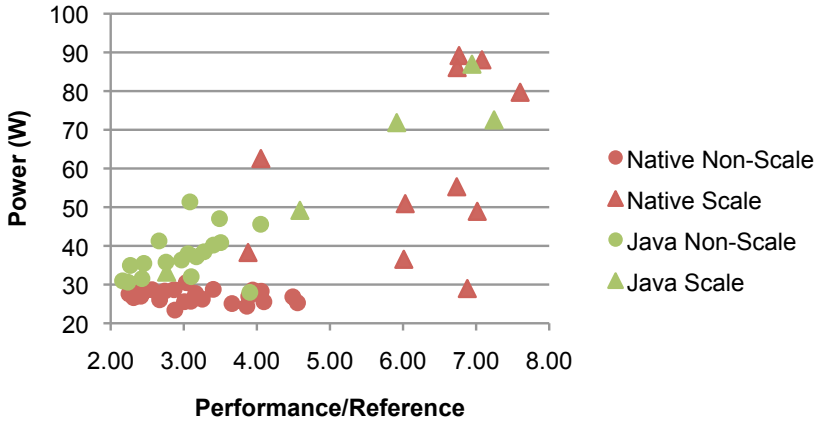


Figure 2.2: Power/performance distribution on the i7 (45). Each point represents one of the 61 benchmarks. Power consumption is highly variable among the benchmarks, spanning from 23W to 89W. The wide spectrum of power responses from different applications points to power saving opportunities in software.

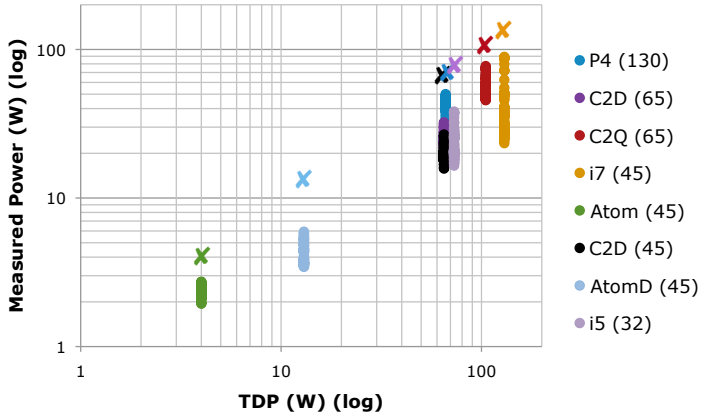


Figure 2.3: Measured power for each processor running 61 benchmarks. Each point represents measured power for one benchmark. The 'X's are the reported TDP for each processor. Power is application-dependent and does not strongly correlate with TDP.

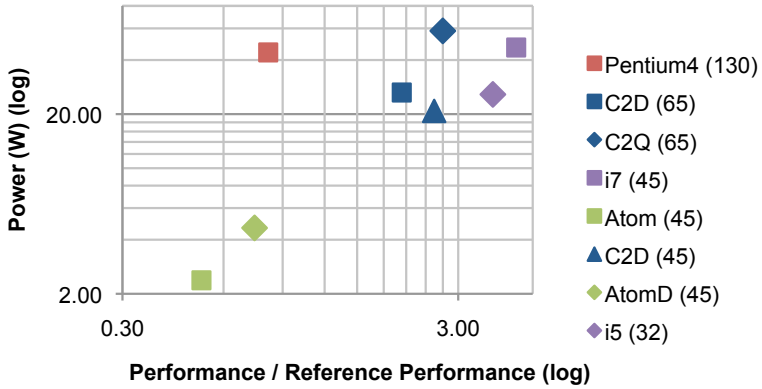
(green) are differentiated by color, whereas scalable (triangle) and non-scalable (circle) are differentiated by shape. Unsurprisingly, the scalable benchmarks (triangles) tend to perform the best and consume the most power. More unexpected is the range of power and performance characteristics of the non-scalable benchmarks. Power is not strongly correlated with performance across workload or benchmarks. The points would form a straight line if the correlation were strong. For example, the point on the bottom right of the figure achieves almost the best relative performance and lowest power.

2.4.2 Historical Overview

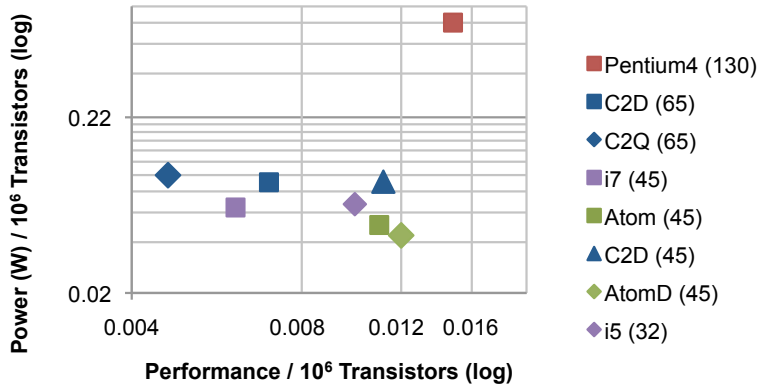
Figure 2.4a plots the average power and performance for each processor in their stock configuration relative to the reference performance, using a log / log scale. For example, the i7 (45) points are the average of the workloads derived from the points in Figure 2.2. Both graphs use the same color for all of the experimental processors in the same family. The shapes encode release age: a square is the oldest, the diamond is next, and the triangle is the youngest, smallest technology in the family.

While historically, mobile devices have been extensively optimized for power, general-purpose processor design has not. Several results stand out that illustrate that power is now a first-order design goal and trumps performance in some cases. (1) The Atom (45) and Atom D (45) are designed as low power processors for a different market, however they successfully execute all these benchmarks and are the most power-efficient processors. Compared to the Pentium 4 (130), they degrade performance modestly and reduce power enormously, consuming as little as one twentieth the power. Device scaling from 130 nm to 45 nm contributes significantly to the power reduction from Pentium to Atom. (2) Comparing between 65 nm and 45 nm generations using the Core 2D (65) and Core 2D (45) shows only a 25% increase in performance, but a 35% drop in power. (3) Comparing the two most recent 45 nm and 32 nm generations using the i7 (45) and i5 (32) shows that the i5 (32) delivers about 15% less performance, while consuming about 40% less power. This result has three root causes: (a) the i7 (45) has four cores instead of two on the i5 (32); (b) since half the benchmarks are scalable multithreaded benchmarks, the software parallelism benefits more from the additional two cores, increasing the advantage to the i7 (45); and (c) the i7 (45) has significantly better memory performance. Comparing the Core 2D (45) to the i5 (32) where the number of processors are matched, the i5 (32) delivers 50% better performance, while consuming around 25% more power than the Core 2D (45).

Contemporaneous comparisons also reveal the tension between power and performance. For example, the contrast between the Core 2D (45) and i7 (45) shows that the i7 (45) delivers 75% more performance than the Core 2D (45), but this performance is very costly in power, with an increase of nearly 100%. These processors thus span a wide range of energy tradeoffs within and across the generations. Overall, these result indicate that optimizing for both power and performance is proving a lot more challenging than optimizing for performance alone.



(a)



(b)

Figure 2.4: Power/performance tradeoff by processor. Each point is an average of the four workloads. (a) Power/performance tradeoffs from Pentium 4 (130) to i5 (32). (b) Power and performance per million transistors. Power per million transistor is consistent across different microarchitectures regardless of the technology node. On average, Intel processors burn around 1 Watt for every 20 million transistors.

Figure 2.4b explores the effect of transistors on power and performance by dividing them by the number of transistors in the *package* for each processor. We include all transistors because our power measurements occur at the level of the package, not the die. This measure is rough and will downplay results for the i5 (32) and Atom D (45), each of which have a GPU in their package. Even though the benchmarks do not exercise the GPUs, we cannot discount them because the GPU transistor counts on the Atom D (45) are undocumented. Note the similarity between the Atom (45), Atom D (45), Core 2D (45), and i5 (32), which at the bottom right of the graph, are the most efficient processors by the transistor metric. Even though the i5 (32) and Core 2D (45) have five to eight times more transistors than the Atom (45), they all eek out very similar performance and power per

transistor. There are likely bigger differences to be found in power efficiency per transistor between chips from different manufactures.

Power per transistor is relatively consistent within microarchitecture family, independent of process technology.

The left-most processors in the graph yield the smallest amount of performance per transistor. Among these processors, the Core 2Q (65) and i7 (45) yield the least performance per transistor and use the largest caches among our set. The large 8 MB caches are not effective. The Pentium 4 (130) is perhaps most remarkable—it yields the most performance per transistor and consumes the most power per transistor by a considerable margin. In summary, performance per transistor is inconsistent across microarchitectures, but power per transistor is more consistent. Power per transistor correlates well with microarchitecture, regardless of technology generation.

2.4.3 Pareto Analysis at 45 nm

The Pareto optimal frontier defines a set of choices that are most efficient in a tradeoff space. Prior research uses the Pareto frontier to explore power versus performance using *models* to derive potential architectural designs on the frontier [3]. We present a Pareto frontier derived from *measured performance and power*. We hold the process technology constant by using the four 45 nm processors: Atom (45), Atom D (45), Core 2D (45), and i7 (45). We expand the number of processor configurations from 4 to 29 by configuring the number of hardware contexts (SMT and CMP), by clock scaling, and disabling / enabling Turbo Boost. The 25 non-stock configurations represent alternative design points. For each configuration, we compute the averages for each workload and their average to produce an energy / performance scatter plot (not shown here). We next pick off the frontier — the points that are not dominated in performance or energy efficiency by any other point — and fit them with a polynomial curve. Figure 2.5 plots these polynomial curves for each workload and the average. The rightmost curve delivers the best performance for the least energy.

Each row of Figure 2.6 corresponds to one of the five curves in Figure 2.5. The check marks identify the Pareto-efficient configurations that define the bounding curve and include 15 of 29 configurations. Somewhat surprising is that none of the Atom D (45) configurations are Pareto ef-

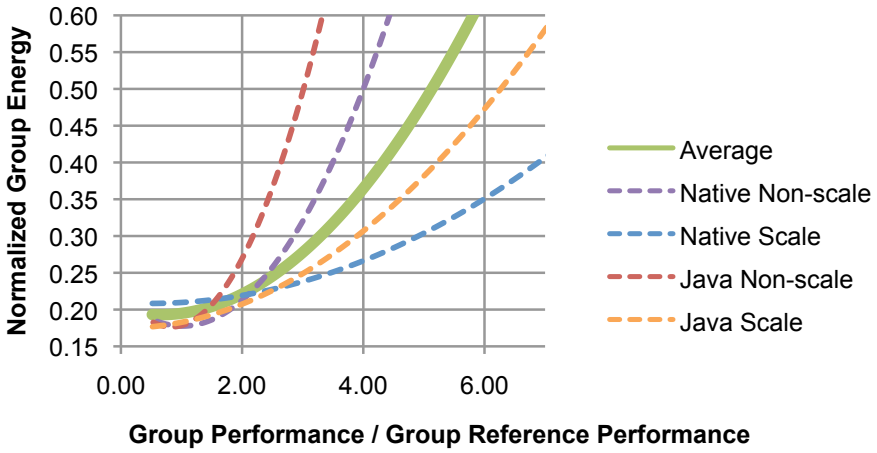


Figure 2.5: Energy / performance Pareto frontiers (45 nm). The energy / performance optimal designs are application-dependent and significantly deviate from the average case.

	Atom (45) 1C2T @ 1.7GHz	Core 2D (45) 2C1T @ 1.6GHz	**Core 2D (45) 2C1T @ 3.1GHz**	i7 (45) 1C1T @ 2.7GHz No TB	i7 (45) 1C1T @ 2.7GHz	i7 (45) 1C2T @ 1.6GHz	i7 (45) 1C2T @ 2.4GHz	i7 (45) 2C1T @ 1.6GHz	i7 (45) 2C2T @ 1.6GHz	i7 (45) 4C1T @ 2.7GHz No TB	i7 (45) 4C1T @ 2.7GHz	i7 (45) 4C1T @ 1.6GHz	i7 (45) 4C2T @ 2.7GHz	i7 (45) 4C2T @ 1.6GHz	i7 (45) 4C2T @ 2.1GHz	i7 (45) 4C2T @ 2.7GHz No TB	**i7 (45) 4C2T @ 2.7GHz**
native non-scalable			✔	✔	✔	✔											
native scalable	✔	✔									✔	✔	✔	✔			
Java non-scalable	✔		✔			✔	✔		✔	✔							✔
Java scalable	✔							✔			✔	✔	✔	✔			
Average	✔							✔			✔	✔	✔	✔			

Figure 2.6: Pareto-efficient processor configurations for each workload. Stock configurations are bold. Each '✔' indicates that the configuration is on the energy/performance Pareto-optimal curve. Native non-scalable has almost no overlap with any other workload.

ficient. Notice the following. (1) Native non-scalable shares only one choice with any other workload. (2) Java scalable and the average share all the same choices. (3) Only two of eleven choices for Java non-scalable and Java scalable are common to both. (4) Native non-scalable does not include the Atom (45) in its frontier. This last finding contradicts prior simulation work, which concluded that dual-issue in-order cores and dual-issue out-of-order cores are Pareto optimal for native non-scalable [3]. Instead we find that all of the Pareto-efficient points for native non-scalable in this design space are quad-issue out-of-order i7 (45) configurations.

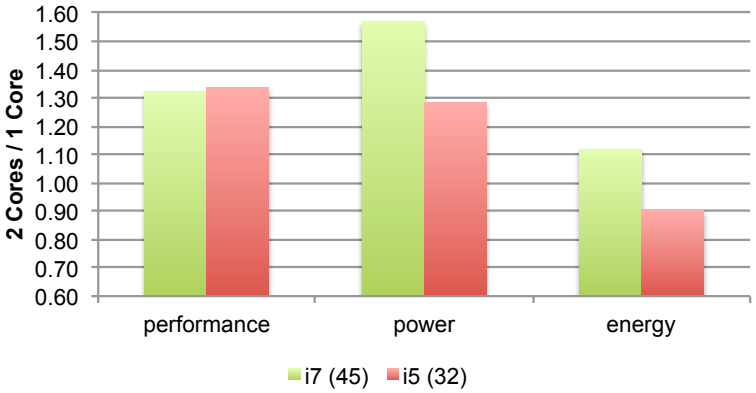
Figure 2.5 starkly shows that each workload deviates substantially from the average. Even when the workloads share points, the points fall in different places on the curves because each workload exhibits a different energy / performance tradeoff. Compare the scalable and non-scalable benchmarks at 0.40 normalized energy on the y-axis. It is impressive how well these architectures effectively exploit software parallelism, pushing the curves to the right and increasing performance from about 3 to 7 while holding energy constant. This measured behavior confirms prior model-based observations about the role of software parallelism in extending the energy / performance curve to the right [3].

Energy-efficient architecture design is very sensitive to workload. Configurations in the native non-scalable Pareto frontier differ substantially from all other workloads.

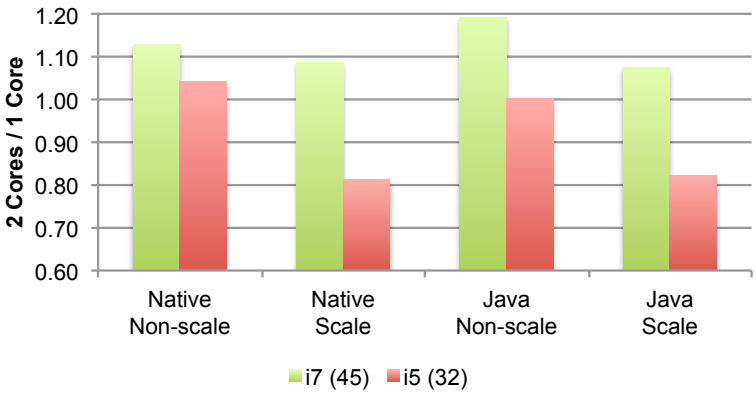
In summary, architects should use a variety of workloads, and in particular, should avoid only using native non-scalable workloads.

2.5 Feature Analysis

Our ASPLOS (2011) paper [33] evaluates the energy effect of a range of hardware features: clock frequency, die shrink, memory hierarchy, hardware parallelism, and gross microarchitecture. This section merely presents results exploring multicore processors (CMP), simultaneous multithreading (SMT), clock scaling, and gross microarchitecture, to give a flavor of our analysis.



(a)



(b)

Figure 2.7: CMP: Comparing two cores to one core. (a) Impact of doubling the number of cores on performance, power, and energy, averaged over all four workloads. (b) Energy impact of doubling the number of cores for each workload. Doubling the cores is not consistently energy efficient among processors or workloads.

2.5.1 Chip Multiprocessors

Figure 2.7 shows the average power, performance, and energy effects of chip multiprocessors (CMPs) by comparing one core to two cores for the two most recent processors in our study. We disable Turbo Boost in these analyses because it adjusts power dynamically based on the number of idle cores. We disable Simultaneous Multithreading (SMT) to maximally expose thread-level parallelism to the CMP hardware feature. Figure 2.7a compares relative power, performance, and energy as a weighted average of the workloads. Figure 2.7b breaks down the energy as a function of workload. While average energy is reduced by 9% when adding a core to the i5 (32), it is increased by 12% when adding a core to the i7 (45). Figure 2.7a shows that the source of this difference is that

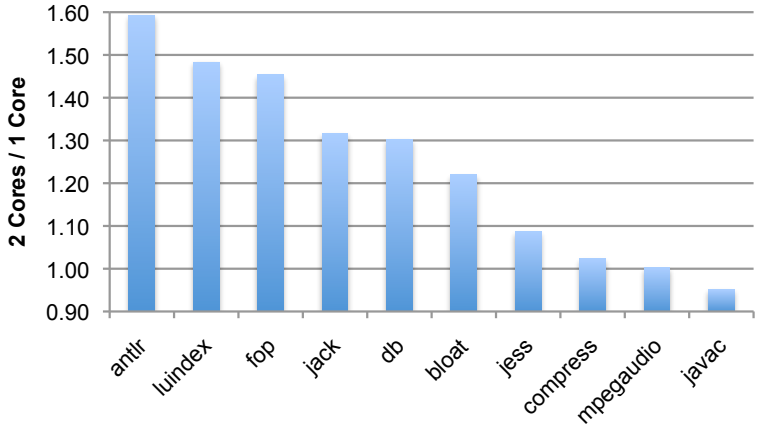


Figure 2.8: Scalability of single-threaded Java benchmarks. Counterintuitively, some single-threaded Java benchmarks scale well. This is because the underlying JVM exploits parallelism for compilation, profiling and garbage collection.

the i7 (45) experiences twice the power overhead for enabling a core as the i5 (32), while producing roughly the same performance improvement.

Comparing one core to two, enabling a core is not consistently energy efficient.

Figure 2.7b shows that native non-scalable and Java non-scalable suffer the most energy overhead with the addition of another core on the i7 (45). As expected, performance for native non-scalable is unaffected. However, turning on an additional core for native non-scalable leads to a power increase of 4% and 14% respectively for the i5 (32) and i7 (45), translating to energy overheads.

More interesting is that Java non-scalable does not incur energy overhead when enabling another core on the i5 (32). In fact, we were surprised to find that the reason for this is that the *single-threaded* Java non-scalable workload runs faster with two processors! Figure 2.8 shows the scalability of the single-threaded subset of Java non-scalable on the i7 (45), with SMT disabled, comparing one and two cores. Although these Java benchmarks are single-threaded, the JVMs on which they execute are not.

The JVM induces parallelism into the execution of single-threaded Java benchmarks.

Since virtual machine runtime services for managed languages, such as just-in-time (JIT) compilation, profiling, and garbage collection, are often concurrent and parallel, they provide substan-

tial scope for parallelization, even within ostensibly sequential applications. We instrumented the HotSpot JVM and found that its JIT compilation and garbage collection are parallel. Detailed performance counter measurements revealed that the garbage collector induced memory system improvements with more cores by reducing the collector's displacement effect on the application thread.

2.5.2 Simultaneous Multithreading

Figure 2.9 shows the effect of disabling simultaneous multithreading (SMT) [111] on the Pentium 4 (130), Atom (45), i5 (32), and i7 (45). Each processor supports two-way SMT. SMT provides fine-grain parallelism to distinct threads in the processors' issue logic and in modern implementations, threads share all processor components (e.g., execution units, caches). Singhal states that the small amount of logic exclusive to SMT consumes very little power [102]. Nonetheless, this logic is integrated, so SMT contributes a small amount to total power even when disabled. Our results therefore slightly underestimate the power cost of SMT. We use only one core, ensuring SMT is the sole opportunity for thread-level parallelism. Figure 2.9a shows that the performance advantage of SMT is significant. Notably, on the i5 (32) and Atom (45), SMT improves average performance significantly without much cost in power, leading to net energy savings.

Simultaneous multithreading delivers substantial energy savings for recent hardware and for in-order processors.

Given that SMT was and continues to be motivated by the challenge of filling issue slots and hiding latency in wide issue superscalars, it may appear counterintuitive that performance on the dual-issue in-order Atom (45) should benefit so much more from SMT than the quad-issue i7 (45) and i5 (32) benefit. One explanation is that the in-order pipelined Atom (45) is more restricted in its capacity to fill issue slots. Compared to other processors in this study, the Atom (45) has much smaller caches. These features accentuate the need to hide latency, and therefore the value of SMT. The performance improvements on the Pentium 4 (130) due to SMT are half to one third that of more recent processors, and consequently there is no net energy advantage. This result is not so surprising given that the Pentium 4 (130) is the first commercial implementation of SMT.

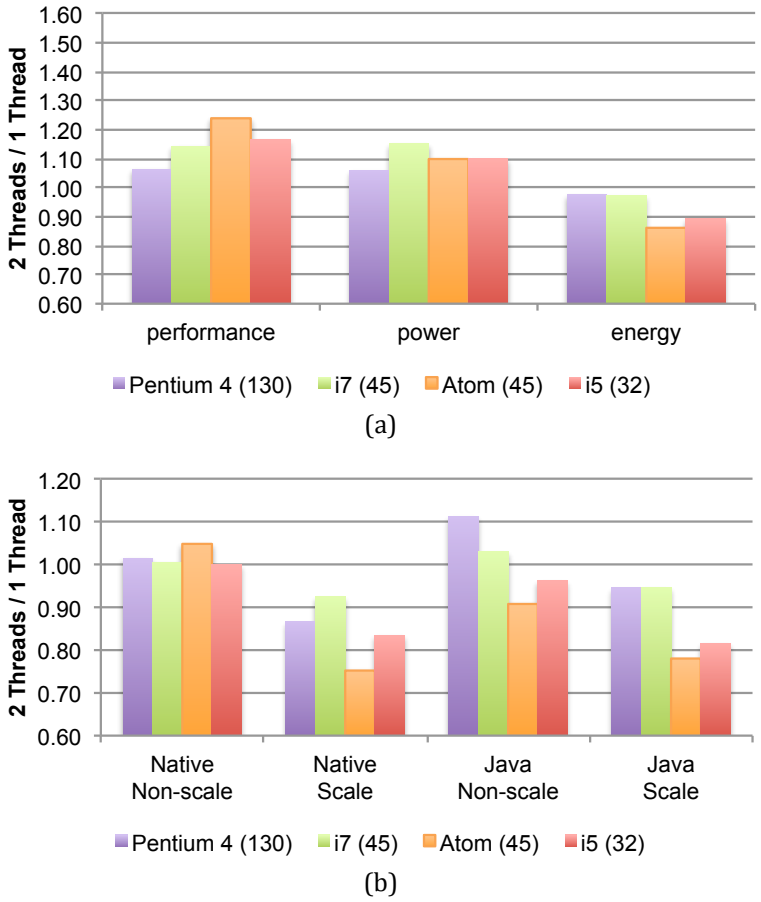
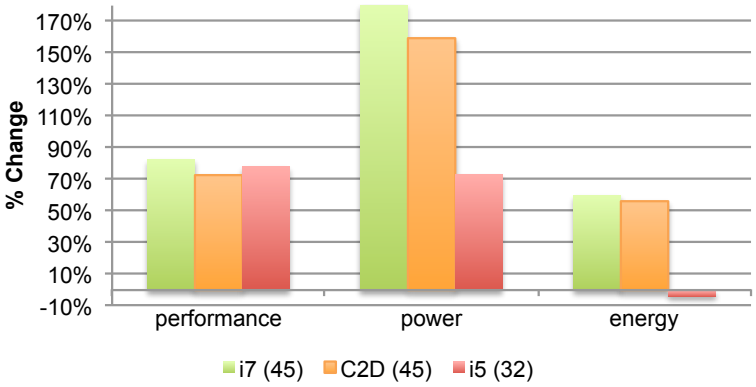


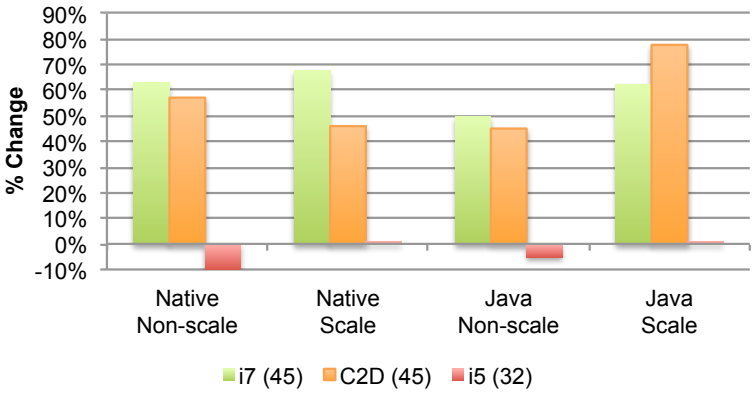
Figure 2.9: SMT: one core with and without SMT. (a) Impact of enabling two-way SMT on a single-core averaged over all four workloads. (b) Energy impact of enabling two-way SMT on a single-core for each workload. Enabling SMT delivers significant energy savings on the recent i5 (32) and the in-order Atom (45).

Figure 2.9b shows that, as expected, the native non-scalable workload experiences very little energy overhead due to enabling SMT, whereas Figure 2.7b shows that enabling a core incurs a significant power and thus energy penalty. The scalable workloads unsurprisingly benefit most from SMT.

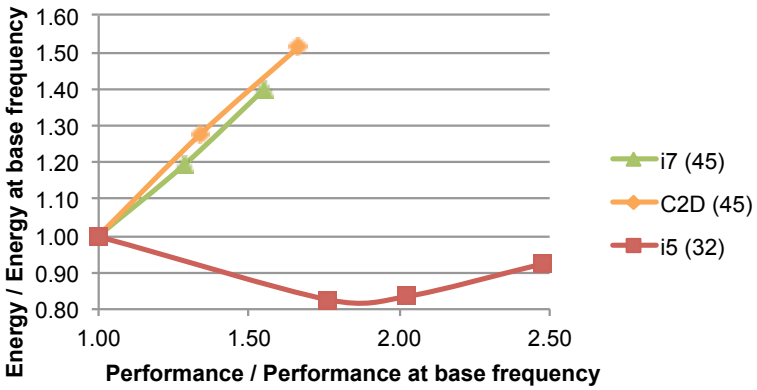
The excellent energy efficiency of SMT is impressive on recent processors as compared to CMP, particularly given its very low die footprint. Compare Figure 2.7 and 2.9. SMT provides less performance improvement than CMP—SMT adds about half as much performance as CMP on average, but incurs much less power cost. The results on the modern processors show SMT in a much more favorable light than in Sasanka et al.'s model-based comparative study of the energy efficiency of SMT and CMP [99].



(a) Average impact of doubling clock.



(b) Workload energy impact of doubling clock.



(c) Average energy performance curve. Points are clock speeds.

Figure 2.10: The impact of clock scaling in stock configurations.

2.5.3 Clock Scaling

We vary the processor clock on the i7 (45), Core 2D (45), and i5 (32) between their minimum and maximum settings. The range of clock speeds are: 1.6 to 2.7 GHz for i7 (45); 1.6 to 3.1GHz for Core 2D (45); and 1.2 to 3.5 GHz for i5 (32). Figures 2.10a and 2.10b express changes in power, performance, and energy with respect to doubling in clock frequency over the range of clock speeds to normalize and compare across architectures.

The three processors experience broadly similar increases in performance of around 80%, but *power differences vary substantially, from 70% to 180%*. On the i7 (45) and Core 2D (45), the performance increases require disproportional power increases—consequently energy consumption increases by about 60% as the clock is doubled. The i5 (32) is starkly different—doubling its clock leads to a slight energy reduction.

The most recent processor in our study does not *consistently increase energy consumption as its clock increases.*

A number of factors may explain why the i5 (32) performs relatively so much better at its highest clock rate: (a) the i5 (32) is a 32 nm process, while the others are 45 nm; (b) the power-performance curve is non-linear and these experiments may observe only the upper (steeper) portion of the curves for i7 (45) and Core 2D (45); (c) although the i5 (32) and i7 (45) share the same microarchitecture, the second generation i5 (32) likely incorporates energy improvements; (d) the i7 (45) is substantially larger than the other processors, with four cores and a larger cache.

2.5.4 Gross Microarchitecture Change

This section explores the power and performance effect of gross microarchitectural change by comparing microarchitectures while matching features such as processor clock, degree of hardware parallelism, process technology, and cache size.

Figure 2.11 compares the Nehalem i7 (45) with the NetBurst Pentium 4 (130), Bonnell Atom D (45), and Core 2D (45) microarchitectures, and it compares the Nehalem i5 (32) with the Core 2D (65). Each comparison configures the Nehalems to match the clock speed, number of cores, and hardware threads of the other architecture. Both the i7 (45) and i5 (32) comparisons to the Core show that

the move from Core to Nehalem yields a small 14% performance improvement. This finding is not inconsistent with Nehalem's stated primary design goals, i.e., delivering scalability and memory performance.

Controlling for technology, hardware parallelism, and clock speed, the out-of-order architectures have similar energy efficiency as the in-order ones.

The comparisons between the i7 (45) and Atom D (45) and Core 2D (45) hold process technology constant at 45 nm. All three processors are remarkably similar in energy consumption. This outcome is all the more interesting because the i7 (45) is disadvantaged since it uses fewer hardware contexts here than in its stock configuration. Furthermore, the i7 (45) integrates more services on-die, such as the memory controller, that are off-die on the other processors, and thus outside the scope of the power meters. The i7 (45) improves upon the Core 2D (45) and Atom D (45) with a more scalable, much higher bandwidth on-chip interconnect, that is not exercised heavily by our workloads. It is impressive that, despite all of these factors, the i7 (45) delivers similar energy efficiency to its two 45 nm peers, particularly when compared to the low-power in-order Atom D (45).

It is unsurprising that the i7 (45) performs 2.6× faster than the Pentium 4 (130), while consuming one third the power, when controlling for clock speed and hardware parallelism (but not for factors such as memory speed). Much of the 50% power improvement is attributable to process technology advances. This speedup of 2.6 over seven years is however *substantially* less than the historical factor of 8 improvement experienced in every prior seven year time interval between 1970 through the early 2000s. This difference in improvements marks the beginning of the power-constrained architecture design era.

2.6 Concluding Remarks

These extensive experiments and analyses yield a wide range of findings. Based on them, we offer the following recommendations in this critical time period of hardware and software upheaval. *Manufacturers* should expose on-chip power meters to the community. *System software and application developers* should understand and optimize power. *Researchers* should use both managed

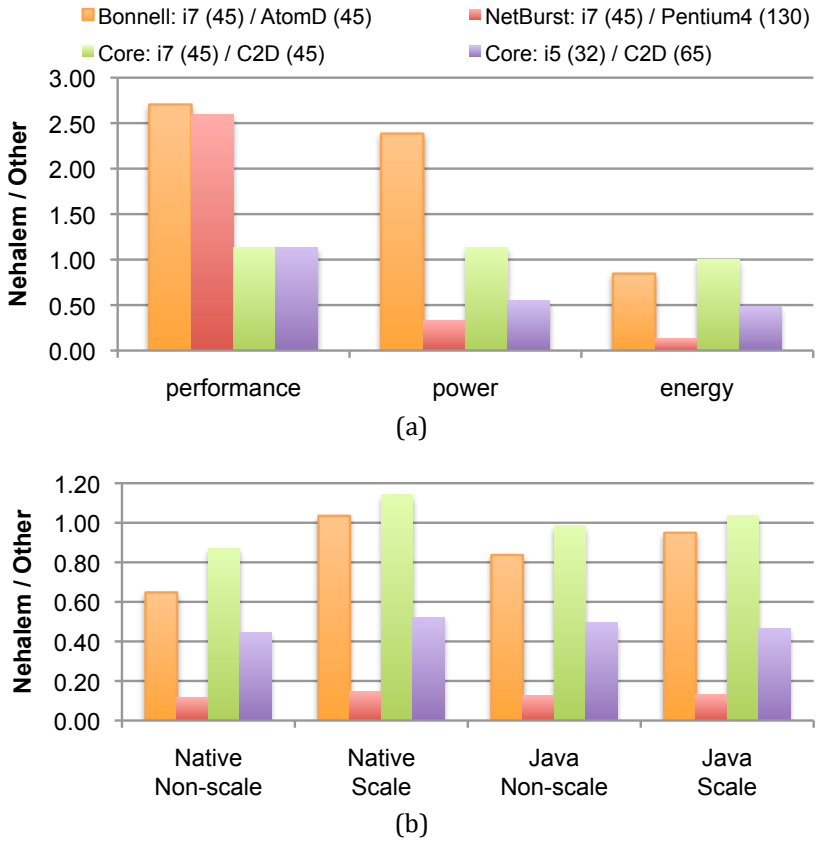


Figure 2.11: Gross microarchitecture: a comparison of Nehalem with four other microarchitectures. In each case, Nehalem is configured to match the other processor as closely as possible. (a) Impact of microarchitecture change with respect to performance, power, and energy, averaged over all four workloads. (b) Energy impact of microarchitecture for each workload. The most recent microarchitecture, Nehalem, is more energy efficient than the others, including the low-power Bonnell (Atom).

and native workloads to quantitatively examine their innovations. *Researchers* should measure power and performance to understand and optimize power, performance, and energy.

CHAPTER 3

DARK SILICON AND THE END OF MULTICORE ERA

Starting in 2004, the microprocessor industry has shifted to multicore scaling–increasing the number of cores per die each generation–as its principal strategy for continuing performance growth. Many in the research community believe that this exponential core scaling will continue into the hundreds or thousands of cores per chip, auguring a parallelism revolution in hardware or software. However, while transistor count increases continue at traditional Moore's Law rates, the per-transistor speed and energy efficiency improvements have slowed dramatically. Under these conditions, more cores are only possible if the cores are slower, simpler, or less utilized with each additional technology generation. This chapter brings together transistor technology, processor core, and application models to understand whether multicore scaling can sustain the historical exponential performance growth in this energy-limited era. As the number of cores increases, power constraints may prevent powering all of the cores at their full speed, requiring a fraction of the cores to be powered off at all times. According to our models, the fraction of these chips that is "dark" may be as much as 50% within three process generations. The low utility of this "dark silicon" may prevent both scaling to higher core counts and ultimately the economic viability of continued silicon scaling. Our results show that core count scaling provides much less performance gain than conventional wisdom suggests. Under (highly) optimistic scaling assumptions–for parallel workloads–*multicore scaling provides a 7.9× (23% per year) over ten years. Under more conservative (realistic) assumptions, multicore scaling provides a total performance gain of 3.7× (14% per year) over ten years, and obviously less when sufficiently parallel workloads are unavailable. Without a breakthrough in process technology or microarchitecture, other directions are needed to continue the historical rate of performance improvement.*

This chapter is based on work presented in ISCA (2011) [32], IEEE Micro Top Picks (2012) [36], ACM Transactions on Computer Systems (2012) [35], and Communications of ACM Research Highlights (2013) [41]. The work is a result of collaboration with Emily Blem[a], Renée St. Amant[b], Karthikeyan Sankaralingam[a], and Doug Burger[c].

[a] University of Wisconsin-Madison

[b] The University of Texas at Austin

[c] Microsoft Research

3.1 Introduction

Moore's Law [84] (the doubling of transistors on chip every 18 months) has been a fundamental driver of computing. For more than four decades, through transistor, circuit, microarchitecture, architecture, and compiler advances, Moore's Law, coupled with Dennard scaling [26], has resulted in consistent exponential performance increases. Dennard's scaling theory showed how to reduce the dimensions and the electrical characteristics of a transistor proportionally to enable successive shrinks that simultaneously improved density, speed, and energy efficiency. According to Dennard's theory with a scaling ratio of $\frac{1}{\sqrt{2}}$, the transistor count doubles (Moore's Law), frequency increases by 40%, and the total chip power stays the same from one generation of process technology to the next on a fixed chip area. With the end of Dennard scaling, process technology scaling can sustain doubling the transistor count every generation, but with significantly less improvement in transistor switching speed and energy efficiency. This transistor scaling trend presages a divergence between energy efficiency gains and transistor density increases. The recent shift to multicore designs, which was partly a response to the end of Dennard scaling, aimed to continue proportional performance scaling by utilizing the increasing transistor count to integrate more cores, which leverage application and/or task parallelism.

Given the transistor scaling trends and challenges, it is timely and crucial for the broader computing community to examine whether multicore scaling will utilize each generation's doubling transistor count effectively to sustain the performance improvements we have come to expect from technology scaling. Even though power and energy have become the primary concern in system design, no one knows how severe (or not) the power problem will be for multicore scaling, especially given the large multicore design space.

Through comprehensive modeling, this chapter provides a decade-long performance scaling projection for future multicore designs. Our multicore modeling takes into account transistor scaling trends, processor core design options, chip multiprocessor organizations, and benchmark characteristics, while applying area and power constraints at future technology nodes. The model combines these factors to project the upper bound speedup achievable through multicore scaling under current technology scaling trends. The model also estimates the effects of nonideal transistor scaling, including the percentage of *dark silicon*–the fraction of the chip that needs to be powered off

at all times–in future multicore chips. Our modeling also discovers the best core organization, the best chip-level topology, and the optimal number of cores for the workloads studied.

The study shows that regardless of chip organization and topology, multicore scaling is power limited to a degree not widely appreciated by the computing community. In just five generations, at 8 nm, the percentage of dark silicon in a fixed-size chip may grow to 50%. Given the recent trend of technology scaling, the 8 nm technology node is expected to be available in 2018. Over this period of ten years (from 2008 when 45 nm microprocessors were available), with optimistic ITRS scaling projections [67], only 7.9× average speedup is possible for commonly used parallel workloads [7], leaving a nearly 24-fold gap from a target of doubled performance per generation. This gap grows to 28-fold with conservative scaling projections [11], with which only 3.7× speedup is achievable in the same period. Further investigations also show that beyond a certain point increasing the core count does not translate to meaningful performance gains. These power and parallelism challenges threaten to end the multicore era, defined as the era during which core counts grow appreciably.

3.2 Overview

Figure 3.1 shows how we build and combine three models to project the performance of future multicores. Ultimately, the model predicts the speedup achievable by multicore scaling and shows a gap between our model's projected speedup and the expected exponential speedup with each technology generation. We refer to this gap as the *dark silicon performance gap*, since it is partly the result of the dark silicon phenomenon, or the nonideal transistor scaling that prevents fully utilizing the exponential increases in transistor count. Our modeling considers transistor scaling projections, single-core design scaling, multicore design choices, application characteristics, and microarchitectural features. This study assumes that the die size and the power budget stays the same as technology scales, an assumption in line with the common practices for microprocessor design. Below we briefly discuss each of the three models.

Device scaling model (M-Device). Two device (transistor) scaling models provide the area, power, and frequency scaling factors at technology nodes from 45 nm through 8 nm. One model is based

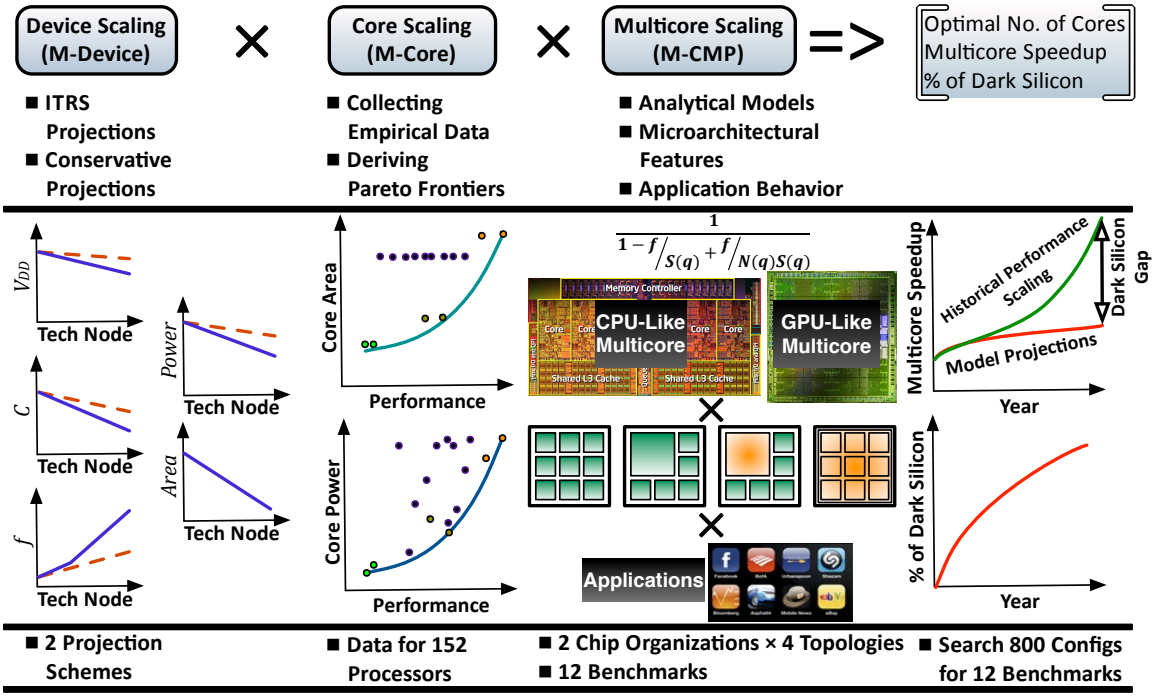


Figure 3.1: Overview of the methodology and the models.

on optimistic ITRS projections [67] while the other one builds on more conservative predictions from Shekhar Borkar's recent study [11].

Core scaling model (M-Core). Through Pareto-optimal curves, derived from measured data, the M-Core model provides the maximum performance that a single-core can sustain for any given area. Further, it provides the minimum power that is consumed to sustain this level of performance. At each technology node, these two Pareto frontiers, which constitute the M-Core model, define the best-case design space of single cores.

Multicore scaling model (M-CMP). The M-CMP models two mainstream classes of multicore organizations, multicore CPUs and many-thread GPUs, which represent two extreme points in the threads-per-core spectrum. The CPU multicore organization represents Intel Nehalem-like multicore designs that benefit from large caches and offer relatively high single-thread performance. The GPU multicore organization represents NVIDIA Tesla-like lightweight cores with heavy multithreading support and poor single-thread performance.

In modeling each of the two multicore organizations, we consider four topologies: symmetric, asymmetric, dynamic, and composed (also called "fused" in the literature [65]). Table 3.1 outlines the four topologies in the multicore design space and the roles of the cores during serial and parallel portions of an application.

Symmetric Multicore. The symmetric, or homogeneous, multicore topology consists of multiple copies of the same core operating at the same voltage and frequency setting. In a symmetric multicore, the resources, including the power and the area budget, are shared equally across all cores.

Asymmetric Multicore. The asymmetric multicore topology consists of one large monolithic core and many identical small cores. This design uses the high-performing large core for the serial portion of code and leverages the numerous small cores as well as the large core to exploit the parallel portion of code.

Dynamic Multicore. The dynamic multicore topology is a variation of the asymmetric multicore topology. During parallel code portions, the large core is shut down and, conversely, during the serial portion, the small cores are turned off and the code runs only on the large core [14, 105]. Switching the cores off and on allows integrating more cores or using a higher voltage and frequency operational setting.

C**omposed Multicore.** The composed multicore topology is a collection of small cores that can dynamically merge together and compose a logical higher performance large core [71, 65]. While providing a parallel substrate for the parallel portion of code when unmerged, the small cores merge and compose a logical core that offers higher single-threaded performance for the serial portion.

The multicore model is an analytic model that computes the multicore performance and takes as input the core performance (obtained from M-Core), the multicore organization (CPU-like or GPU-like), and multicore topology (symmetric, asymmetric, dynamic, and composed). Unlike previous studies, the model also takes into account application characteristics such as memory access pattern, and the amount of thread-level parallelism in the workload as well as the microarchitectural features such as cache size and memory bandwidth. We choose the PARSEC benchmarks [7]

Table 3.1: CPU and GPU topologies. Single-thread (ST) cores are uni-processor style cores with large caches and many-thread (MT) cores are GPU-style cores with smaller caches.

	Code	Symmetric	Asymmetric	Dynamic	Composed
CPU Multicores	Serial Fraction	1 ST Core	1 Large ST Core	1 Large ST Core	1 Large ST Core
	Parallel Fraction	N ST Cores	1 Large ST Core + N Small ST Cores	N Small ST Cores	N Small ST Cores
GPU Multicores	Serial Fraction	1 MT Core (1 Thread)	1 Large ST Core (1 Thread)	1 Large ST Core (1 Thread)	1 Large ST Core (1 Thread)
	Parallel Fraction	N MT Cores (Multiple Threads)	1 Large ST Core + N Small MT Cores (1 Thread) (Multiple Threads)	N Small MT Cores (Multiple Threads)	N Small MT Cores (Multiple Threads)

to study the multicore scaling potential for successfully parallelized applications. PARSEC is a set of highly parallel applications that are widely used to support the parallel architecture research.

Modeling future multicore chips. To model future multicore chips, we first model the building blocks, the future cores. We combine the device and core models to project the best-case design space of single cores—the Pareto frontiers—at future technology nodes. Any performance improvement for future cores will come at the cost of area or power as defined by the projected Pareto frontiers. Then, we combine all three models and perform an exhaustive design-space search to find the optimal multicore configuration for each individual application considering its characteristics. The optimal configuration delivers the maximum multicore speedup for each benchmark at future technology nodes while enforcing area and power constraints. The gap between the projected speedup and the speedup we have come to expect with each technology generation is the dark silicon performance gap.

Modeling heterogenous multicores. Heterogeneous configurations such as AMD Fusion and Intel Sandy Bridge combine CPU and GPU designs on a single chip. The asymmetric and dynamic GPU topologies resemble those two designs, and the composed topology models configurations similar to AMD Bulldozer. For GPU-like multicores, this study assumes that the single ST core does not participate in parallel work. Finally, our methodology implicitly models heterogeneous cores of

different types (mix of issue widths, frequencies, etc.) integrated on one chip. Since we perform a per-benchmark optimal search for each organization and topology, we implicitly cover the upper-bound of this heterogeneous case.

When modeling multicores, previous work largely abstracts away processor organization and application details. However, this study provides a comprehensive multicore modeling framework that considers the implications of process technology scaling, decouples power and area constraints, uses real measurements to model single-core design tradeoffs, exhaustively considers multicore design space and its various organizations, incorporates microarchitectural features, and examines real applications and their characteristics.

3.3 Device Model (M-Device)

The first step in projecting gains from more cores is developing a model that captures future transistor scaling trends. To highlight the challenges of nonideal device scaling, first we present a simplified overview of historical Dennard scaling and the more recent scaling trends.

Historical device scaling trends: According to Dennard scaling, as the geometric dimensions of transistors scale, the electric field within the transistors stays constant if other physical features, such as the gate oxide thickness and doping concentrations, are reduced proportionally. To keep the electric field constant, the supply voltage (the switch on voltage) as well as the threshold voltage (the voltage level below which the transistor switches off) need to be scaled at the same rate as the dimensions of the transistor. With Dennard scaling, a 30% reduction in transistor length and width results in a 50% decrease in transistor area, doubling the number of transistors that can fit on chip with each technology generation (Moore's Law [84]). Furthermore, the decrease in transistor sizes results in a 30% reduction in delay. In total, Dennard scaling suggests a 30% reduction in delay (hence 40% increase in frequency), a 50% reduction in area, and a 50% reduction in power per transistor. As a result, the chip power stays the same as the number of transistors doubles from one technology node to the next in the same area.

Recent device scaling trends: At recent technology nodes, the rate of supply voltage scaling has dramatically slowed due to limits in threshold voltage scaling. Leakage current increases expo-

nentially when the threshold voltage is decreased, limiting threshold voltage scaling, and making leakage power a significant and first-order constraint. Additionally, as technology scales to smaller nodes, physics limits decreases in gate oxide thickness. These two phenomena were not considered in the original Dennard scaling theory, since leakage power was not dominant in the older generations, and the physical limits of scaling oxide thickness were too far out to be considered. Consequently, Dennard scaling stopped at 90 nm [27]. That is, transistor area continues to scale at the historic rate, which allows for doubling the number of transistors, while the power per transistor is not scaling at the same rate. This disparity will translate to an increase in chip power if the fraction of active transistors is not reduced from one technology generation to the next [14]. The shift to multicore architectures was partly a response to the end of Dennard scaling.

3.3.1 Model Structure

The device model provides transistor area, power, and frequency scaling factors from a base technology node (e.g. 45 nm) to future technologies. The area scaling factor corresponds to the shrinkage in transistor dimensions. The frequency scaling factor is calculated based on the fan-out of 4 (FO4) delay reduction. FO4 is a process independent delay metric used to measure the delay of CMOS logic that identifies the processor frequency. FO4 is the delay of an inverter, driven by an inverter 4$\times$ smaller than itself, and driving an inverter 4$\times$ larger than itself. The power scaling factor is computed using the predicted frequency, voltage, and gate capacitance scaling factors in accordance with the $P = \alpha C V_{dd}^2 f$ equation.

3.3.2 Model Implementation

To avoid any bias in our studies, we build four device models. Two *original* models, conservative and ITRS, and two *synthetic* models derived from the original ones, midpoint and aggressive. The parameters used for calculating the power and performance scaling factors are summarized in Table 3.2.

Original M-Device models. The conservative model is based on predictions presented by Borkar that takes a more measured approach in predicting the physical properties of the transistors and

Table 3.2: Scaling factors for conservative, midpoint, ITRS and aggressive projections.

	Tech Node (nm)	Frequency Scaling Factor (/45 nm)	Vdd Scaling Factor (/45 nm)	Capacitance Scaling Factor (/45 nm)	Power Scaling Factor (/45 nm)	Energy Scaling Factor (/45 nm)	Area Scaling Factor (/45 nm)
Conservative	45	1.00	1.00	1.00	1.00	1.00	2^{0}
	32	1.10	0.93	0.75	0.71	0.65	2^{-1}
	22	1.19	0.88	0.56	0.52	0.44	2^{-2}
	16	1.25	0.86	0.42	0.39	0.31	2^{-3}
	11	1.30	0.84	0.32	0.29	0.22	2^{-4}
	8	1.34	0.84	0.24	0.22	0.16	2^{-5}
6% frequency increase, 23% power reduction, and 30% energy reduction per node							
Midpoint	45	1.00	–	–	1.00	1.00	2^{0}
	32	1.10	–	–	0.69	0.63	2^{-1}
	22	1.79	–	–	0.53	0.30	2^{-2}
	16	2.23	–	–	0.39	0.17	2^{-3}
	11	2.74	–	–	0.27	0.10	2^{-4}
	8	2.60	–	–	0.17	0.07	2^{-5}
21% frequency increase, 30% power reduction, and 42% energy reduction per node							
ITRS	45*	1.00	1.00	1.00	1.00	1.00	2^{0}
	32*	1.09	0.93	0.70	0.66	0.61	2^{-1}
	22†	2.38	0.84	0.33	0.54	0.23	2^{-2}
	16†	3.21	0.75	0.21	0.38	0.12	2^{-3}
	11†	4.17	0.68	0.13	0.25	0.06	2^{-4}
	8†	3.85	0.62	0.08	0.12	0.03	2^{-5}
31% frequency increase, 35% power reduction, and 50% energy reduction per node							
Aggressive	45	1.00	–	–	1.00	1.00	2^{0}
	32	1.11	–	–	0.64	0.57	2^{-1}
	22	2.98	–	–	0.51	0.17	2^{-2}
	16	4.19	–	–	0.38	0.09	2^{-3}
	11	5.61	–	–	0.23	0.04	2^{-4}
	8	5.11	–	–	0.07	0.01	2^{-5}
39% frequency increase and 41% power reduction, and 58% energy reduction per node							

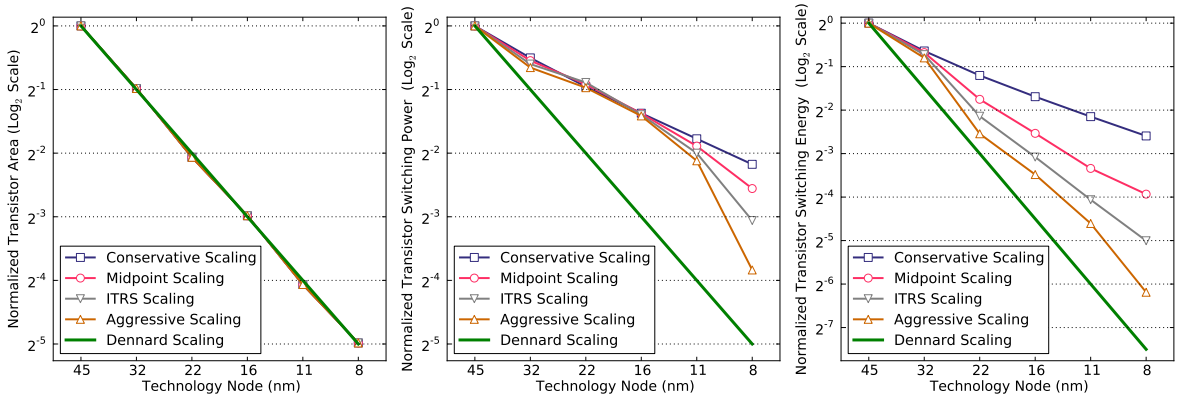
$*$: Extended Planar Bulk Transistors, †:Multi-Gate Transistors

represents a less optimistic view [11]. The ITRS model uses projections from the ITRS 2010 technology roadmap [67]. ITRS is an industry consortium that sets targets and goal for the microelectronics industry. As shown in Table 3.2, the ITRS roadmap predicts that multi-gate MOSFETs, such as FinFETs, will supersede planar bulk at 22 nm [67]. The large increase in frequency, 2.2× as shown in Table 3.2, and substantial reduction in capacitance, 47%, from 32 nm to 22 nm is the result of this technology change. The ITRS roadmap predicts that by changing the transistor technology to multi-gate MOSFETs, the device power decreases by 18%, despite a frequency increase of 2.2×. Based on ITRS projections, the device switching delay increases from 11 nm to 8 nm, while its power decreases.

Synthetic M-Device models. The midpoint model is the middle ground scaling projection between conservative scaling and the ITRS projection. At each technology node, the frequency scaling and the power scaling factors are computed as the average of conservative and ITRS factors. For the aggressive model, which is one step more optimistic than ITRS, the frequency and power scaling factors are computed such that the ITRS factors are the average of midpoint and aggressive factors. In all the four models, the energy scaling factor is computed based on the frequency and power factors as the (power scaling factor) × 1/(frequency scaling factor). The area scaling factor is the same across all models: a 50% area reduction per process scaling in accordance with Moore's Law [84].

Figure 3.2 shows the device area, switching power, and switching energy scaling trends for our four M-Device models across the future technology nodes compared to classical Dennard scaling. As illustrated, even though the device area is scaling according to historical rates, there is a growing gap between device power and the historical rate of transistor power reduction. This growing gap is one of the main sources of dark silicon. At 8 nm, the gap is 2.3× between Dennard scaling and the device switching power projected by the aggressive model. This gap becomes as large as 7.3× with conservative scaling at 8 nm.

Leakage. We allocate 20% of the chip power budget to leakage power. As shown in [89], the transistor threshold voltage can be selected so that the maximum leakage power is always an acceptable ratio of the chip power budget while still meeting power and performance constraints.



(a) Device area (b) Device switching power (c) Device switching energy

Figure 3.2: Device scaling trends across future technology nodes with four different scaling projections compared to classical Dennard scaling.

We also observe that with 10% or 30% leakage power, we do not see significant changes in optimal configurations.

3.4 Core Model (M-Core)

The second step in estimating future multicore performance is modeling a key building block, the processor core.

3.4.1 Model Structure

We build the technology-scalable core model by populating the area/performance and power/performance design spaces with the data collected for a set of processors as depicted in Figure 3.3. The core model is the combination of the area/performance Pareto frontier, $A(q)$, and the power/performance Pareto frontier, $P(q)$, for these two design spaces. The q is the single-threaded performance of a core. These frontiers capture the optimal area/performance and power/performance trade-offs for a core while abstracting away specific details of the core. We use the device scaling model to project the frontiers to future technologies and model performance, area, and power of cores fabricated at those nodes.

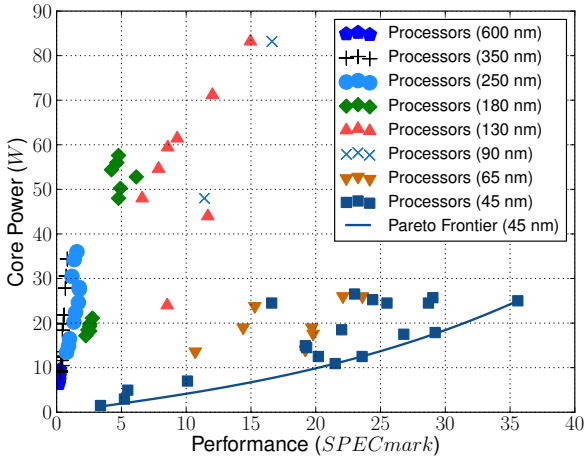
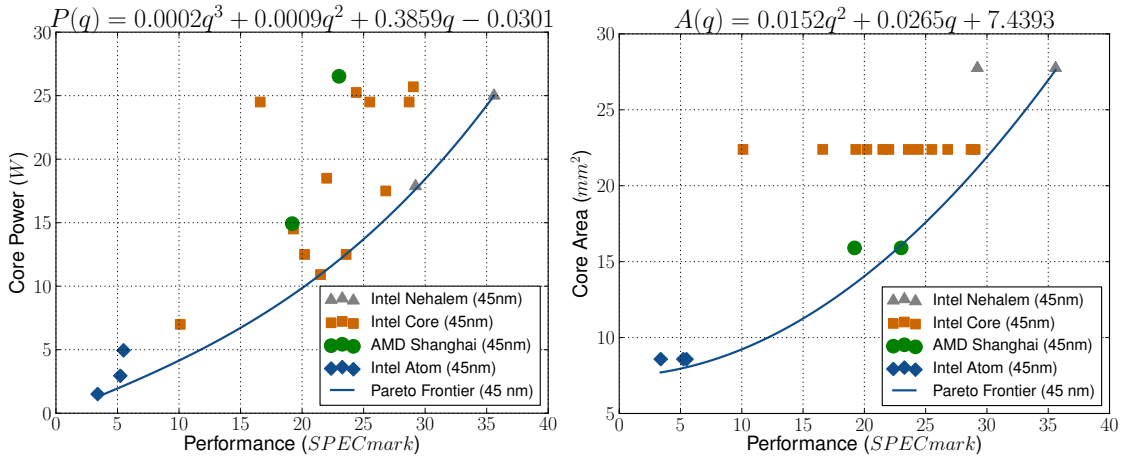


Figure 3.3: Power/performance design space of 152 processors (from P54C Pentium to Nehalem-based i7) fabricated at 600 nm through 45 nm. The design space boundary that comprises the optimal points constructs the Pareto frontier.



(a) Power/performance frontier, 45 nm

(b) Area/performance frontier, 45 nm

Figure 3.4: Single-core (a) power/performance and (b) area/performance design space at 45 nm and the corresponding Pareto frontiers.

3.4.2 Model Implementation

As Figure 3.4 depicts, we populate the two design spaces at 45 nm using 20 representative Intel and AMD processors[1] and derive the Pareto frontiers. The curve that bounds all power(area)/performance points in the design space and minimizes power(area) required for a given level of performance constructs the Pareto frontier. The polynomials $P(q)$ and $A(q)$ are the *core model*. The core performance, q, is the processor's SPECmark and is collected from the SPEC website [103]. We estimate the core power budget using the TDP reported in processor datasheets. TDP is the chip power budget, or the amount of power the chip can dissipate without exceeding the transistor junction temperature. We used die photos of the four microarchitectures, Intel Atom, Intel Core, AMD Shanghai, and Intel Nehalem, to estimate the core areas (excluding level 2 and level 3 caches). Since the focus of this work is to study the impact of technology constraints on logic scaling rather than cache scaling, we derive the Pareto frontiers using only the portion of *power budget* and area allocated to the core in each processor excluding the 'uncore' components. To compute the power budget of a single core, the power budget allocated to the level 2 and level 3 caches is estimated and deducted from the chip TDP. In the case of a multicore CPU, the remainder of the chip power budget is divided by the number of cores, resulting in the power budget allocated to a single core

As illustrated in Figure 3.4, a cubic polynomial, $P(q)$, is fit to the points along the edge of the power/performance design space and a quadratic polynomial (Pollack's rule [92]), $A(q)$, to the points along the edge of the area/performance design space. We used the least square regression method for curve fitting such that the frontiers enclose all design points. Figures 3.4a and 3.4b show the 45 nm processor points and identify the power/performance and area/performance Pareto frontiers. The power/performance cubic polynomial $P(q)$ function (Figure 3.4a) and the area/performance quadratic polynomial $A(q)$ (Figure 3.4b) are the *core model*. The Intel Atom Z520 with an estimated 1.89 W core TDP represents the lowest power design (lower-left frontier point), and the Nehalem-based Intel Core i7-965 Extreme Edition with an estimated 31.25 W core TDP represents the highest performing design (upper-right frontier point). The points along the scaled Pareto

[1] Atom Z520, Atom 230, Atom D510, Core 2 Duo T9500, Core 2 Extreme QX9650, Core 2 Quad Q8400, Opteron 2393SE, Opteron 2381HE, Core 2 Duo E7600, Core 2 Duo E8600, Core 2 Quad Q9650, Core 2 Quad QX9770, Core 2 Duo T9900, Pentium SU2700, Xeon E5405, Xeon E5205, Xeon X3440, Xeon E7450, Core i7-965 ExEd

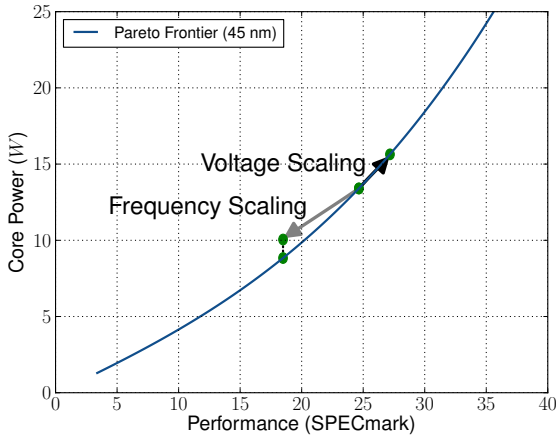


Figure 3.5: Voltage and frequency scaling on the power/performance Pareto frontiers.

frontier are used as the search space for determining the best core configuration by the multicore-scaling model.

Decoupling power/performance trade-off from area/performance trade-off. Previous studies on multicore performance modeling [62, 18, 81, 74, 115, 14, 19] use Pollack's rule [92] to denote the trade-off between area and performance. Furthermore, these studies consider the power consumption of a core to be directly proportional to its area. This assumption makes power an area-dependent constraint. However, power is a function of not only area, but also supply voltage and frequency. Since these no longer scale at historical rates, Pollack's rule is insufficient for modeling core power. Thus, it is necessary to decouple area and power into two independent constraints.

Voltage and frequency scaling. Our device and core models do not explicitly consider dynamic voltage and frequency scaling; instead, we take an optimistic approach to account for its best-case impact. When deriving the Pareto frontiers, each processor data point was assumed to operate at its optimal voltage and frequency setting $(Vdd_{min}, Freq_{max})$. Figure 3.5 shows the result of voltage/frequency scaling on the design points along the power/performance frontier. At a fixed Vdd setting, scaling down the frequency from $Freq_{max}$ results in a power/performance point inside the optimal Pareto curve, a suboptimal design point. However, scaling voltage up and operating at a new $(Vdd'_{min}, Freq'_{max})$ setting results in a different power-performance point that is still on the optimal frontier. Furthermore, if an application dissipates less than the power budget, we assume

Table 3.3: M-CMP parameters with default values from 45 nm Nehalem.

Parameter	Description	Default	Affected By
N	Number of cores	4	Multicore Topology
T	Number of threads per core	1	Core Style
$freq$	Core frequency (MHz)	3200	Core Performance
CPI_{exe}	Cycles per instruction (zero-latency cache accesses)	1	Core Performance, Application
C_{L1}	L1 cache size per core (KB)	64	Core Style
C_{L2}	L2 cache size per chip (MB)	2	Core Style, Multicore Topology
t_{L1}	L1 access time (cycles)	3	-
t_{L2}	L2 access time (cycles)	20	-
t_{mem}	Memory access time (cycles)	426	Core Performance
BW_{max}	Maximum memory bandwidth (GB/s)	200	Technology Node
b	Bytes per memory access (B)	64	-
f	Fraction of code that can be parallel	varies	Application
r_m	Fraction of instructions that are memory accesses	varies	Application
α_{L1}, β_{L1}	L1 cache miss rate function constants	varies	Application
α_{L2}, β_{L2}	L2 cache miss rate function constants	varies	Application

that the voltage and frequency scaling will be utilized to achieve the highest possible performance with the minimum power increase. This is possible since voltage and frequency scaling only changes the operating condition in a Pareto-optimal fashion. Since we investigate all of the points along the frontier to find the optimal multicore configuration, our study covers multicore designs that induce heterogeneity to symmetric topologies through dynamic voltage and frequency scaling.

3.5 Multicore Model (M-CMP)

The last step in modeling multicore scaling is to develop a detailed chip-level model (M-CMP) that integrates the area and power frontiers, microarchitectural features, and application behavior, while accounting for the chip organization (CPU-like or GPU-like) and its topology (symmetric, asymmetric, dynamic, or composed).

3.5.1 Model Structure

Guz et al. proposed a model to consider first-order impacts of microarchitectural features (cache organization, memory bandwidth, number of threads per core, etc.) and workload characteristics (memory access pattern) [53]. To first order, their model considers stalls due to memory dependences and resource constraints (bandwidth or functional units). We extend their approach to build our multicore model. Our extensions incorporate additional application characteristics, microar-

chitectural features, and physical constraints, and covers both homogeneous and heterogeneous multicore topologies.

This model uses single-threaded cores with large caches to cover the CPU multicore design space and massively threaded cores with minimal caches to cover the GPU multicore design while modeling all four topologies. The input parameters to the model, and how, if at all, they are affected by the multicore design choices are listed in Table 3.3.

Multicore topologies. The multicore model is an extended Amdahl's Law [2] equation that incorporates the multicore performance ($Perf$) calculated from (3.2)-(3.5):

$$Speedup = 1/\left(\frac{f}{S_{Parallel}} + \frac{1-f}{S_{Serial}}\right) \tag{3.1}$$

The M-CMP model (3.1) measures the multicore speedup with respect to a baseline multicore ($Perf_B$). That is, the parallel portion of code (f) is sped up by $S_{Parallel} = Perf_P/Perf_B$ and the serial portion of code ($1-f$) is sped up by $S_{Serial} = Perf_S/Perf_B$.

The number of cores that fit on the chip is calculated as follows based on the topology of the multicore, its area budget ($AREA$), its power budget (TDP), each core's area ($A(q)$), and each core's power ($P(q)$).

$$\begin{aligned}
N_{Symm}(q) &= \min\left(\frac{AREA}{A(q)}, \frac{TDP}{P(q)}\right) \\
N_{Asym}(q_L, q_S) &= \min\left(\frac{AREA - A(q_L)}{A(q_S)}, \frac{TDP - P(q_L)}{P(q_S)}\right) \\
N_{dynm}(q_L, q_S) &= \min\left(\frac{AREA - A(q_L)}{A(q_S)}, \frac{TDP}{P(q_S)}\right) \\
N_{Comp}(q_L, q_S) &= \min\left(\frac{AREA}{(1+\tau)A(q_S)}, \frac{TDP}{P(q_S)}\right)
\end{aligned}$$

For heterogeneous multicores, q_S is the single-threaded performance of the small cores and q_L is the single-threaded performance of the large core. The area overhead of supporting composability is τ; however, no power overhead is assumed for composability support.

Microarchitectural features. Multithreaded performance ($Perf$) of an either CPU-like or GPU-like multicore running a fully parallel ($f = 1$) and multithreaded application is calculated in terms of instructions per second in (3.2) by multiplying the number of cores (N) by the core utilization (η) and scaling by the ratio of the processor frequency to CPI_{exe}:

$$Perf = \text{min}\left(N\frac{freq}{CPI_{exe}}\eta, \frac{BW_{max}}{r_m \times m_{L1} \times m_{L2} \times b}\right) \tag{3.2}$$

The CPI_{exe} parameter does not include stalls due to cache accesses, which are considered separately in the core utilization (η). The core utilization is the fraction of time that a thread running on the core can keep it busy. It is modeled as a function of the average time spent waiting for each memory access (t), fraction of instructions that access the memory (r_m), and the CPI_{exe}:

$$\eta = \text{min}\left(1, \frac{T}{1 + t\frac{r_m}{CPI_{exe}}}\right) \tag{3.3}$$

The average time spent waiting for memory accesses (t) is a function of the time to access the caches (t_{L1} and t_{L2}), time to visit memory (t_{mem}), and the predicted cache miss rate (m_{L1} and m_{L2}):

$$t = (1 - m_{L1})t_{L1} + m_{L1}(1 - m_{L2})t_{L2} + m_{L1}m_{L2}t_{mem} \tag{3.4}$$

$$m_{L1} = \left(\frac{C_{L1}}{T\beta_{L1}}\right)^{1-\alpha_{L1}} \text{ and } m_{L2} = \left(\frac{C_{L2}}{NT\beta_{L2}}\right)^{1-\alpha_{L2}} \tag{3.5}$$

3.5.2 Model Implementation

The M-CMP model incorporates the Pareto frontiers, physical constraints, real application characteristics, and realistic microarchitectural features into the multicore speedup projections as discussed below.

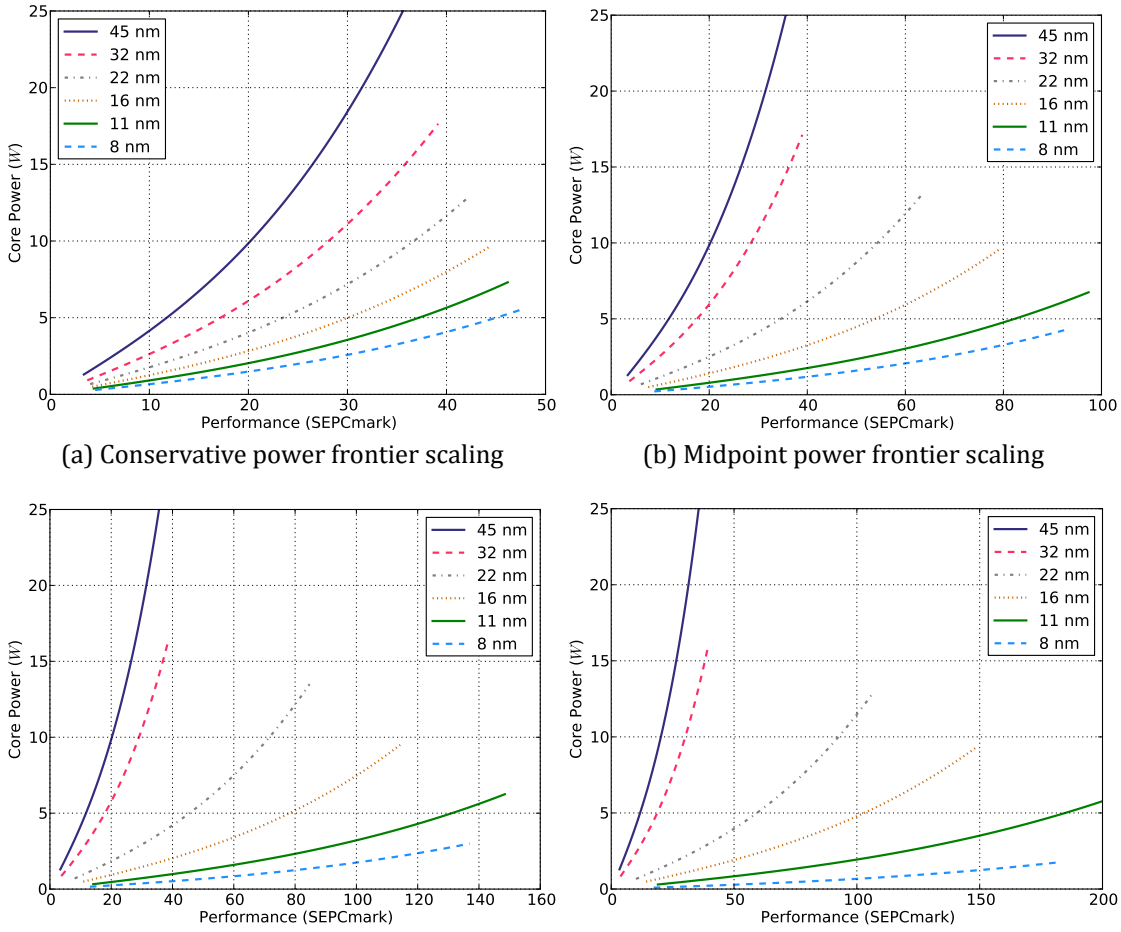
Application characteristics. The input parameters that characterize an application are its cache behavior, fraction of instructions that are loads or stores, and fraction of parallel code. For the PARSEC benchmarks, we obtain this data from two previous studies [6, 7]. To obtain the fraction of parallel code (f) for each benchmark, we fit an Amdahl's Law-based curve to the reported speedups

across different numbers of cores from both studies. The value of f ranges from 0.75 to 0.9999 depending on the benchmark.

Obtaining frequency and CPI_{exe} from Pareto frontiers. To incorporate the Pareto-optimal curves into the M-CMP model, we convert the SPECmark scores (q) into an estimated CPI_{exe} and core frequency. We assume the core frequency scales linearly with performance, from 1.5 GHz for an Atom core to 3.2 GHz for a Nehalem core. Each application's CPI_{exe} is dependent on its instruction mix and use of hardware resources (e.g., functional units and out-of-order issue width). Since the measured CPI_{exe} for each benchmark at each technology node is not available, we use the M-CMP model to generate per benchmark CPI_{exe} estimates for each design point along the Pareto frontier. With all other model inputs kept constant, we iteratively search for the CPI_{exe} at each processor design point. We start by assuming that the Nehalem core has a CPI_{exe} of ℓ. Then, the smallest core, an Atom processor, should have a CPI_{exe} such that the ratio of its M-CMP performance to the Nehalem core's M-CMP performance is the same as the ratio of their SPECmark scores (q). We assume CPI_{exe} does not change as technology scales, while frequency does change as discussed in Section 3.6.1.

Microarchitectural features. A key part of the detailed model is the set of input parameters that model the microarchitecture of the cores. For single-thread (ST) cores, we assume each core has a 64 KB L1 cache, and chips with only ST cores have an L2 cache that is 30% of the chip area. Many-thread (MT) cores have small L1 caches (32 KB for every 8 cores), support multiple hardware contexts (1024 threads per 8 cores), a thread register file, and no L2 cache. From Atom and Tesla die photos, we estimate that 8 small MT cores, their shared L1 cache, and their thread register file can fit in the same area as one Atom processor. We assume that off-chip bandwidth (BW_{max}) increases linearly as process technology scales down while the memory access time is constant.

Composed multicores. We assume that τ (area overhead of composability) increases from 10% up to 400% depending on the total area of the composed core and performance of the composed core cannot exceed performance of a single Nehalem core at 45 nm.



(a) Conservative power frontier scaling

(b) Midpoint power frontier scaling

(c) ITRS power frontier scaling

(d) Aggressive power frontier scaling

Figure 3.6: Combining M-Device and M-Core results in core models for future technology nodes, the scaled power/performance and area/performance Pareto frontier pairs.

Constraints and baseline. The area and power budgets are derived from the highest-performing quad-core Nehalem multicore at 45 nm excluding the L2 and L3 caches. They are 111 mm^2 and 125 W, respectively. The M-CMP multicore speedup baseline is a quad-Nehalem multicore that fits in the area and power budgets. The reported dark silicon projections are for the area budget that is solely allocated to the cores, not caches and other 'uncore' components. The actual fraction of chip that goes dark may be higher.

3.6 Combining Models

3.6.1 Device × Core Model

To study core scaling in future technology nodes, we scaled the 45 nm Pareto frontiers down to 8 nm by scaling the power and performance of each processor data point using the $DevM$ model and then re-fitting the Pareto optimal curves at each technology node. Performance, measured in SPECmark, is assumed to scale linearly with frequency. This optimistic assumption ignores the effects of memory latency and bandwidth on the core performance, and thus actual performance gains through scaling may be lower. Figures 3.6 shows the scaled power Pareto frontiers with the conservative, midpoint, ITRS, and aggressive device scaling models. As illustrated in Figure 3.6(a), conservative scaling suggests that performance will increase only by 34%, and power will decrease by 74% as a core scales from 45 nm to 8 nm. Figure 3.6(b) shows that the core performance from 45 nm through 8 nm increases 2.6× and the core power decreases by 83% with midpoint scaling. Based on the optimistic ITRS predictions, however, scaling a microarchitecture (core) from 45 nm to 8 nm will result in a 3.9× performance improvement and an 88% reduction in its power consumption (Figure 3.6(c)). As shown in Figures 3.6(d), with aggressive scaling the single-threaded performance at 8 nm increases by a factor of 5.1× while its power dissipation decreases by 93%. The current trends of frequency scaling in microprocessor design is far from the predictions of ITRS. We believe that based on the current trends, without any disruptive innovations in transistor design, such frequency improvements may not be possible.

3.6.2 Device × Core × Multicore Model

All three models are combined to produce final projections on optimal multicore speedup, optimal number of cores, and amount of dark silicon. To determine the best multicore configuration at each technology node, we sweep the design points along the scaled area/performance and power/performance Pareto frontiers (M-Device × M-Core) as these points represent the most efficient designs. At each technology node, for each core design on the scaled frontiers, we construct a multicore chip consisting of one such core. For a symmetric multicore chip, we iteratively add identical cores one by one until the area or power budget is hit, or performance improvement is limited. We

sweep the frontier and construct a symmetric multicore for each processor design point. From this set of symmetric multicores, we pick the multicore with the best speedup as the optimal symmetric multicore for that technology node. The procedure is similar for other topologies. This procedure is performed separately for CPU-like and GPU-like organizations. The amount of dark silicon is the difference between the area occupied by cores for the optimal multicore and the area budget allocated to the cores.

3.7 Scaling and Future Multicores

We apply the combined models to study the future of multicore designs and their performance limiting factors. The results from this study provide detailed analysis of multicore behavior for future technologies considering 12 real applications from the PARSEC suite. Details for all applications and topologies are presented in Figure 3.17. Unless otherwise stated, the model is used to find the optimal multicore configuration with the highest possible speedup for each individual benchmark.

3.7.1 Speedup Projections

Figure 3.7 shows the geometric mean of speedup and the best-case speedup among the benchmarks for a symmetric topology using the optimistic ITRS scaling. The symmetric topology achieves the lower bound on speedups. With speedups that are no more than 10% higher, the dynamic and composed topologies achieve the upper-bound. The results are presented for both CPU-like and GPU-like multicore organizations. To conduct a fair comparison between different design points, all speedup results are normalized to the performance of a quad-core Nehalem multicore at 45 nm that fits in the same power and area budget. The results over five technology generations with the four device scaling projections are summarized below:

		Conservative		**Midpoint**		**ITRS**		**Aggressive**	
	Topology	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
Geomean Speedup	Symmetric	3.4×	2.4×	5.4×	2.4×	7.7×	2.7×	10.2×	2.7×
Geomean Speedup	Asymmetric	3.5×	2.4×	5.5×	2.4×	7.9×	2.7×	10.3×	2.7×
Geomean Speedup	Dynamic	3.5×	2.4×	5.5×	2.4×	7.9×	2.7×	10.3×	2.7×
Geomean Speedup	Composed	3.7×	2.3×	5.1×	2.3×	6.2×	2.5×	7.2×	2.5×
Maximum Speedup	All	10.9×	10.1×	27.5×	10.1×	46.6×	11.2×	91.7×	11.2×

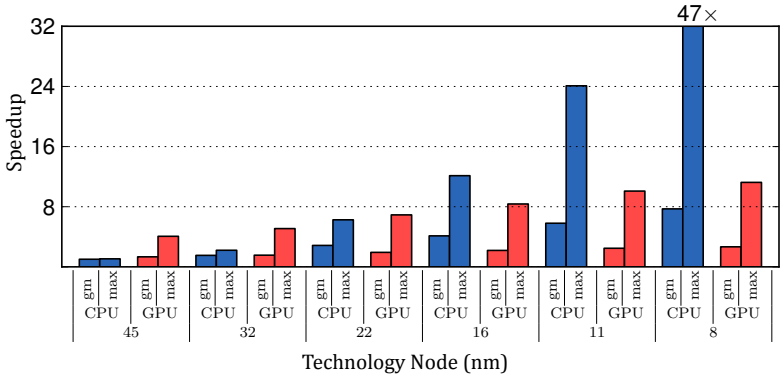


Figure 3.7: Speedup projections for CPU-like and GPU-like symmetric multicore topology across technology generations with ITRS scaling.

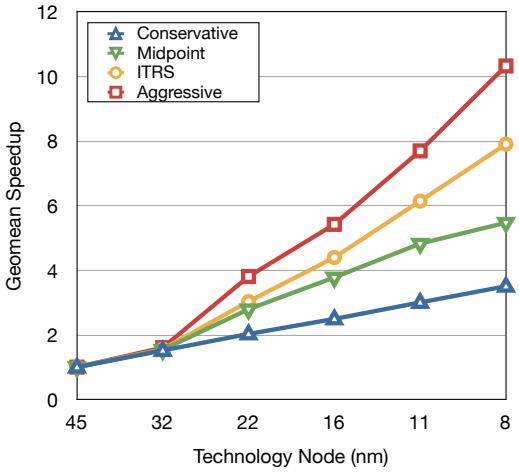
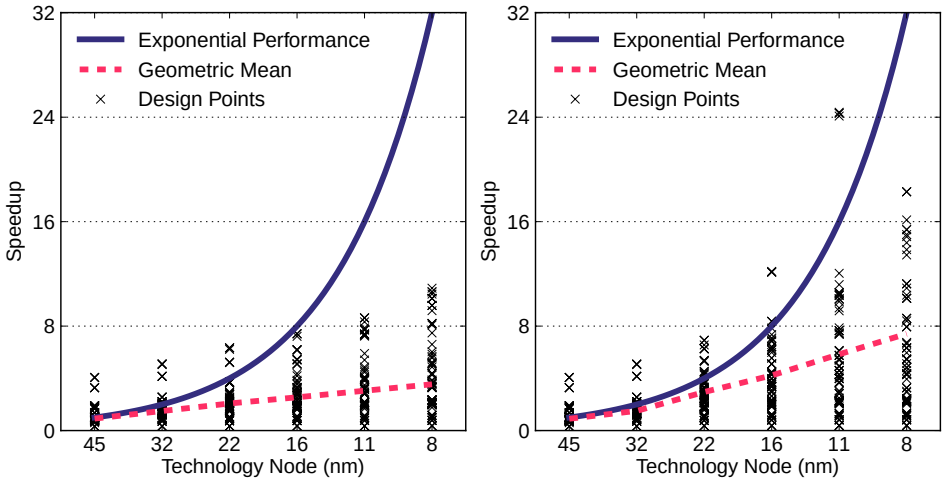


Figure 3.8: Speedup projections for dynamic CPU-like multicore topologies with four transistor scaling models.

Figure 3.8 shows the geometric mean speedup projections across the technology nodes for all the device scaling models. As depicted, improvements in process technology have a direct impact in bridging the dark silicon speedup gap. We believe that reality will be closer to the midpoint projections that leaves a large dark silicon speedup gap. However, a disruptive breakthrough in transistor fabrication that matches the aggressive scaling predictions could improve potential multicore scaling significantly.

Figure 3.9 summarizes all of the speedup projections in a single scatter plot for conservative and ITRS scaling models. For every benchmark at each technology node, we plot the speedup of eight possible multicore configurations (CPU-like, GPU-like) $\times$ (symmetric, asymmetric, dynamic, com-



(a) Conservative Scaling (b) ITRS Scaling

Figure 3.9: Speedup across process technology nodes across all organizations and topologies with PARSEC benchmarks.

posed). The upper curve indicates *performance* Moore's Law or doubling performance for every technology generation.

With optimal multicore configurations for each individual application, at 8 nm, only 3.7× (conservative scaling), 5.5× (midpoint scaling), 7.9× (ITRS scaling), or 10.3× (aggressive scaling) geometric mean speedup is possible.

Highly parallel workloads with a degree of parallelism higher than 99% will continue to benefit from multicore scaling.

At 8 nm, the geometric mean speedup for heterogeneous dynamic and composed topologies is only 10% higher than the geometric mean speedup for symmetric topologies.

Improvements in transistor process technology are directly reflected as multicore speedup; however, to bridge the dark silicon speedup gap even a disruptive breakthrough that matches our aggressive scaling model is not enough.

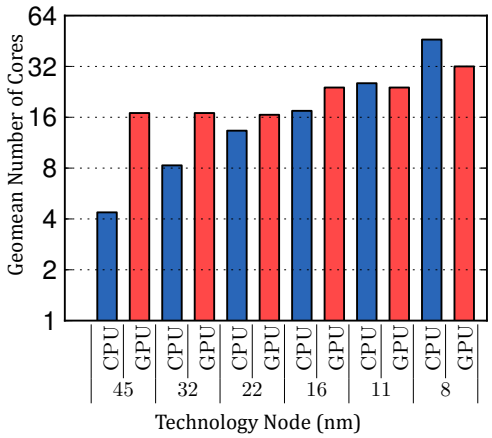
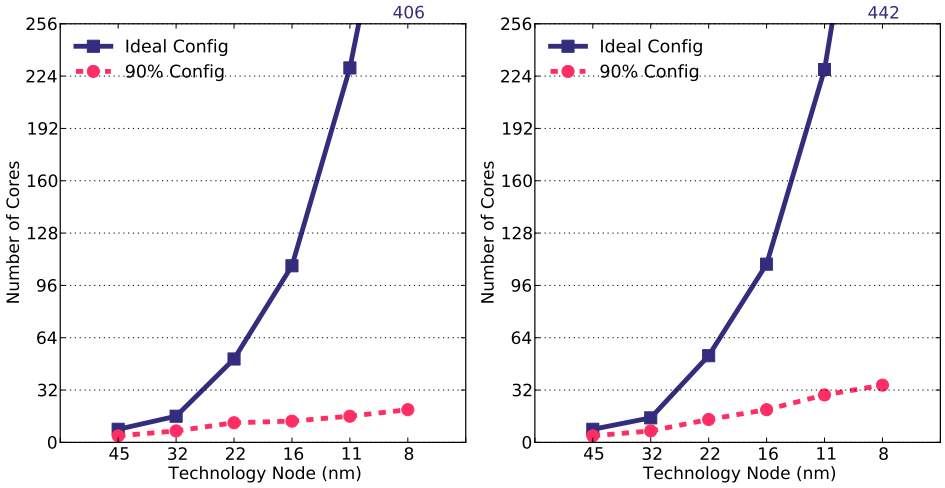


Figure 3.10: Geometric mean number of cores across 12 PARSEC benchmarks for symmetric topology with ITRS scaling.

3.7.2 Core Count Projections

Figure 3.10 illustrates the geometric mean number of cores across the 12 PARSEC benchmarks for a symmetric topology with ITRS scaling when each individual benchmark has its optimum number of cores. Different applications saturate performance improvements at different core counts, but the geometric mean number of cores is less than 64. We consider as an *ideal* configuration the chip configuration that provides the best speedups for all applications. Figure 3.11 shows the number of cores (solid line) for the ideal CPU-like dynamic multicore configuration across technology generations, since dynamic configurations performed best. The dashed line illustrates the number of cores required to achieve 90% of the ideal configuration's geometric mean speedup across PARSEC benchmarks. As depicted, with ITRS scaling, the ideal configuration integrates 442 cores at 8 nm; however, 35 cores reach the 90% of the speedup achievable by 442 cores. With conservative scaling, the 90% speedup core count is 20 at 8 nm.

For the PARSEC benchmarks that we studied, the typical number of cores for individual benchmarks is less than 64 for CPUs and less than 256 SP cores for GPUs with both conservative and ITRS scaling.



(a) Conservative Scaling (b) ITRS Scaling

Figure 3.11: Number of cores for the ideal CPU-like dynamic multicore configurations and the number of cores delivering 90% of the speedup achievable by the ideal configurations across the PARSEC benchmarks.

Due to limited parallelism in the PARSEC benchmark suite, even with novel heterogeneous topologies and optimistic ITRS scaling, integrating more than 35 cores improves performance only slightly for CPU-like topologies.

The optimal number of cores projected by our study seems small compared to chips such as the NVIDIA Fermi, which has 512 cores at 45 nm. There are two reasons for this discrepancy. First, in our study we are optimizing for a fixed power budget, whereas with real GPUs the power has been slightly increasing. Second, our study optimizes core count and multicore configuration for general purpose workloads similar to the PARSEC suite. We assume Fermi is optimized for graphics rendering. When we applied our methodology to a graphics kernel (ray tracing) in an asymmetric topology, we obtained higher speedups and an optimal core count of 4864 at 8 nm, with 8% dark silicon.

3.7.3 Dark Silicon Projections

Figure 3.13 illustrates the dark silicon projections for dynamic CPU multicore topology with the four transistor scaling models. As depicted, the midpoint scaling almost matches the ITRS projections and aggressive scaling lowers the portion of dark silicon. However, as shown in Figure 3.2, even with

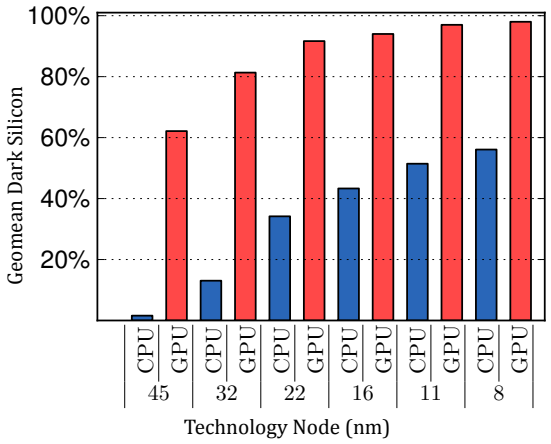


Figure 3.12: Percentage of dark silicon (geometric mean across all 12 PARSEC benchmarks) for symmetric topology and ITRS scaling.

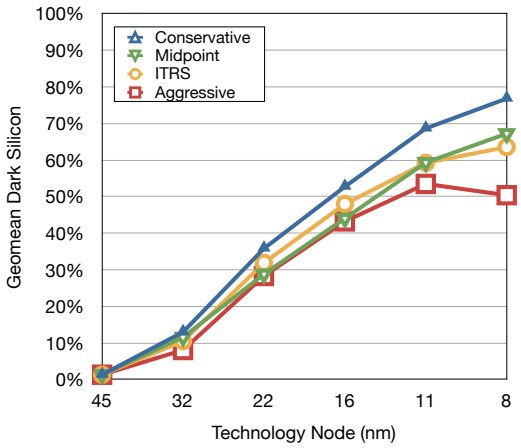


Figure 3.13: Dark silicon projections across technology generation for dynamic CPU-like multicores with the four device scaling models.

aggressive scaling, the energy efficiency of transistors is significantly below the ideal Dennard scaling which enables Moore's Law. This significant gap between the ideal transistor scaling prevents even huge improvements in process technology from bridging the dark silicon *underutilization* gap. Microarchitectural innovations that can efficiently trade area for energy are vital to tackle the dark silicon problem.

Figure 3.12 depicts the geometric mean percentage of dark silicon across the PARSEC benchmarks for symmetric multicores with ITRS scaling. In these projections, we optimistically use the core counts that achieve the highest speedup for individual benchmarks. The trend is similar for other topologies.

With conservative scaling, dark silicon dominates in 2016 for CPU-like and in 2012 for GPU-like multicores. With ITRS scaling, dark silicon dominates in 2021 for CPU-like multicores and in 2015 for GPU-like multicores.

With ITRS projections, at 22 nm (2012) 21% of the chip will be dark and at 8 nm, over 50% of the chip cannot be utilized.

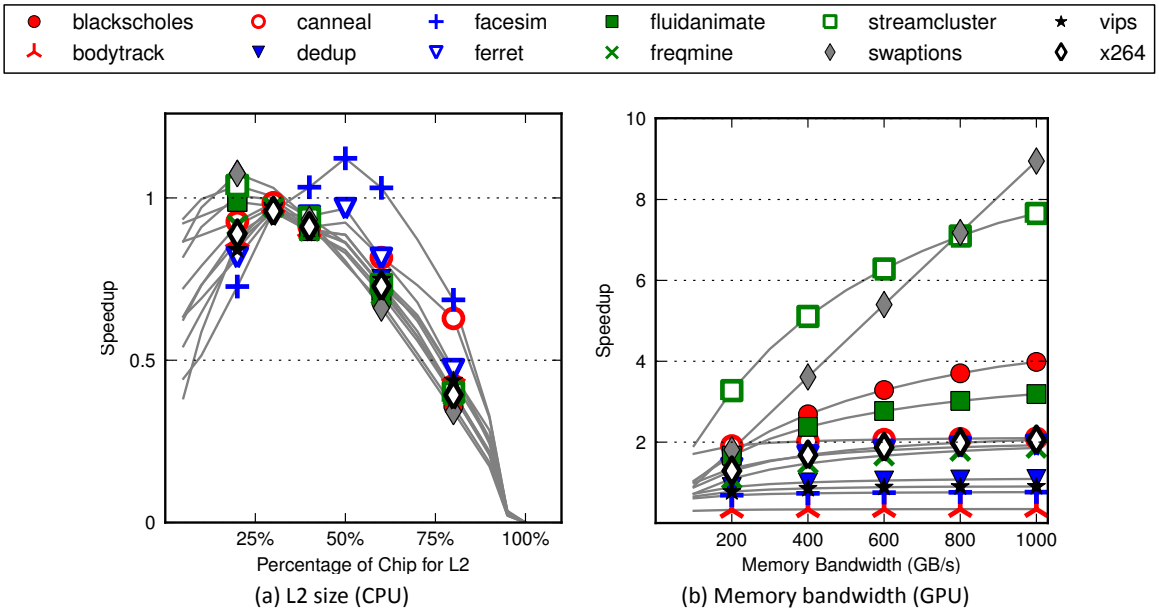


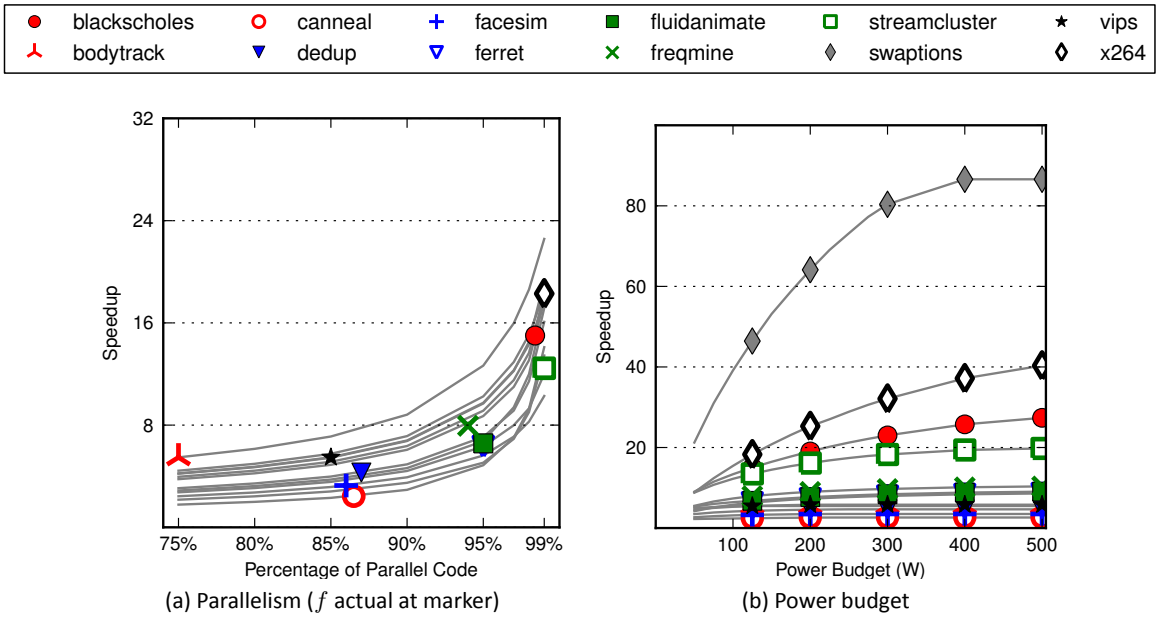
Figure 3.14: Impact of L2 size and memory bandwidth on speedup at 45 nm.

Mere improvements in process technology even as significant as aggressive scaling projections cannot bridge the dark silicon underutilization gap. Microarchitecture innovations are vital to justify continuous scaling.

3.7.4 Bridging the Dark Silicon Gap

Our analysis thus far examined "typical" configurations and showed poor scalability for the multicore approach. A natural question is, *can simple configuration changes (percentage cache area, memory bandwidth, etc.) provide significant benefits?* We elaborate on three representative studies of simple changes (L2 cache size, memory bandwidth, and SMT) below. Further, to understand whether parallelism or the power budget is the primary source of the dark silicon speedup gap, we vary each of these factors in two experiments at 8 nm. Our model is flexible enough to perform these types of studies.

L2 cache area. Figure 3.14(a) shows the optimal speedup at 45 nm as the amount of a symmetric CPU's chip area devoted to L2 cache varies from 0% to 100%. In this study we ignore any increase in L2 cache power or increase in L2 cache access latency. Across the PARSEC benchmarks, the optimal percentage of chip devoted to cache varies from 20% to 50% depending on benchmark memory



(a) Parallelism (f actual at marker) (b) Power budget

Figure 3.15: Impact of application parallelism and power budget on speedup at 8 nm.

access characteristics. Compared to a 30% cache area, using optimal cache area only improves performance by at most 20% across all benchmarks.

Memory bandwidth. Figure 3.14(b) illustrates the sensitivity of PARSEC performance to the available memory bandwidth for symmetric GPU multicores at 45 nm. As the memory bandwidth increases, the speedup improves as the bandwidth can keep more threads fed with data; however, the increases are limited by power and/or parallelism and in 10 out of 12 benchmarks speedups do not increase by more than 2× compared to the baseline, 200 GB/s.

SMT. To simplify the discussion, we did not consider SMT support for the processors (cores) in the CPU multicore organization. SMT support can improve the power efficiency of the cores for parallel workloads to some extent. We studied 2-way, 4-way, and 8-way SMT with no area or energy penalty, and observed that speedup improves with 2-way SMT by 1.5× in the best case and decreases as much as 0.6× in the worst case due to increased cache contention; the range for 8-way SMT is 0.3-2.5×.

Application parallelism. First, we keep the power budget constant (our default budget is 125 W), and vary the level of parallelism in the PARSEC applications from 0.75 to 0.99, assuming programmer effort can realize this improvement. We see performance improves slowly as the parallelism level

increases, with most benchmarks reaching a speedup of about only 15× at 99% parallelism. Provided that the power budget is the only limiting factor, typical upper-bound ITRS-scaling speedups would still be limited to 15×. With conservative scaling, this best-case speedup is limited to 6.3×.

Power-budget/lower-power cores. For the second experiment, we keep each application's parallelism at its real level and vary the power budget from 50 W to 500 W. Eight of 12 benchmarks show no more than 10× speedup even with a practically unlimited power budget. That is, increasing core counts beyond a certain point does not improve performance due to the limited parallelism in the applications and Amdahl's Law. Only four benchmarks have sufficient parallelism to even hypothetically sustain Moore's Law level speedups.

The level of parallelism in PARSEC applications is the primary contributor to the dark silicon speedup gap. However, in realistic settings the dark silicon resulting from power constraints limits the achievable speedup.

3.8 Model Assumptions, Validation, and Limitations

We elaborate on the assumptions of the model and through validation against empirical results demonstrate that they are carefully considered and consistently optimistic with respect to the multicore speedup projections. In addition, our modeling includes certain limitations, which we argue they do not significantly change the results.

3.8.1 Model Assumptions

The M-CMP model allows us to estimate the first-order impact of caching, parallelism, and threading under several key assumptions. Table 3.4 qualitatively describes the impact of these assumptions. The model optimistically assumes that the workload is homogeneous, work is infinitely parallel during parallel sections of code, and no thread synchronization, operating system serialization, or swapping occurs. We also assume memory accesses never stall due to a previous access. Each of these assumptions results in over-prediction of multicore performance, making the model and hence projected speedups optimistic. Cache behaviors may lead to over- or under-prediction. The model assumes that each thread only sees its own slice of cache and thus the model may over or

Table 3.4: Effect of assumptions on M-CMP accuracy. Assumptions lead to ↑ (slightly higher), ⇑ (higher) or ↓ (slightly lower) predicted speedups (or have no effect (—)).

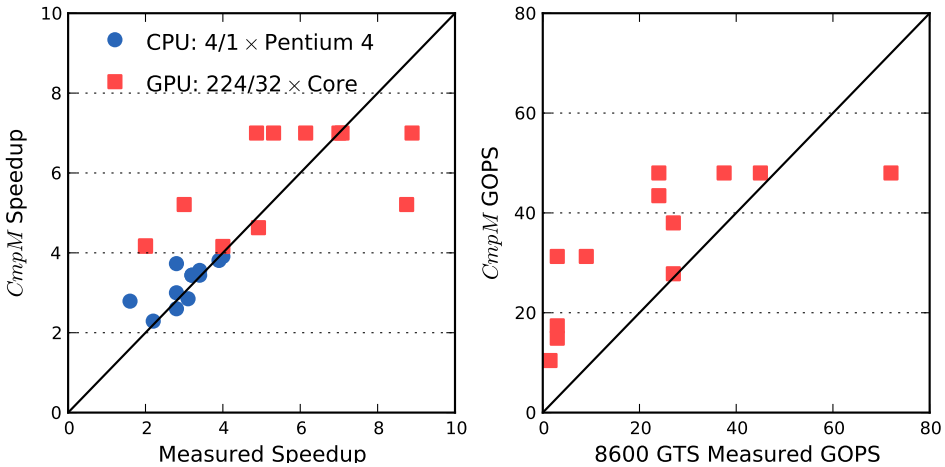
	Assumption	Impact on CPU Speed	Impact on GPU Speed
μarch	Memory Contention: 0	↑	⇑
	Interconnection Network Latency: 0	⇑	⇑
	Thread Swap Time: 0	↑	↑
Application	Cache Hit Rate Function	↑ or ↓	↑ or ↓
	Thread Synch & Communication: 0	↑	⇑
	Thread Data Sharing: 0	↓	—
	Workload Type: Homogeneous	↑	↑

underestimate the hit rate. However, comparing the model's output to the published empirical results confirms that it only over-predicts multicore performance.

3.8.2 Model Validation

To validate the M-CMP model, we compare the speedup projections from the model to measurement and simulation results for existing CPU and GPU multicores. For the CPU case, we compare the model's speedup predictions to speedup measurements for a quad-Pentium 4 multicore [6]. The model is configured to match this real multicore. We validate GPU speedup projections by comparing the model's output simulation results from GPGPUSim [5]. Both model and simulator compare speedups of a 224-core GPU over a 32-core GPU. We use GPGPUSim's 12 CUDA benchmarks since GPU implementations of PARSEC are not available. Figure 3.16a, which includes both CPU and GPU data, shows that the model is optimistic and over predicts the speedups. M-CMP underpredicts speedups for two benchmarks for which the simulation results show a speedup of greater than $7\times$, the increase in number of cores.

To strongly advance our GPU claim, we also need to prove the model's raw performance projection is accurate or optimistic. As depicted in Figure 3.16b, the model's GPU performance projection is validated by comparing its output to the results from a real system, NVIDIA 8600 GTS, using the data from [5]. Except for a known anomaly that also occurs in GPGPUsim, M-CMP consistently over-predicts raw performance.



(a) Speedup (b) Performance

Figure 3.16: M-CMP model validation.

Furthermore, using our model, we find $4\times$ geometric-mean and $12\times$ maximum speedup for PARSEC benchmarks on Tesla compared to a quad-core Nehalem. While our results are impressively close to Intel's empirical measurements using similar benchmarks [75], the match in the model's maximum speedup prediction ($12\times$ vs $11\times$ in the Intel study) is an anomaly. Our model does not account for specialized compute units, which contribute to the speedup in [75].

3.8.3 Model Limitations

Different workloads. Workloads with significantly different behavior than the workloads we studied could result in different findings.

Other types of cores. We do not consider embedded ARM or Tilera cores in this work because they are designed for different application domains and their SPECmark scores are not available for a meaningful comparison.

Power impact of uncore. We ignore the power impact of uncore components such as the memory subsystem. There is consensus that the number of these components will increase and hence they will further eat into the power budget, reducing speedups.

GPU methodology. Our GPU methodology may over-estimate the GPU power budget, so we investigated the impact of 10%-50% improved energy efficiency for GPUs and found that total chip speedup and percentage of dark silicon were not impacted.

We acknowledge that we make a number of assumptions in this work to build a useful model. Questions may still linger on the model's accuracy and whether its assumptions contribute to the performance projections that fall well below the ideal 32×. First, in all instances, we selected parameter values that would be favorable towards multicore performance. Second, our validation against real and simulated systems shows the model always over-predicts multicore performance.

3.9 Concluding Remarks

For decades, Dennard scaling permitted more, faster, *and* more energy efficient transistors with each new process technology, justifying the enormous costs required to develop each new process node. Dennard scaling's failure led the industry to race down the multicore path, which for some time permitted performance scaling for parallel and multitasked workloads, permitting the economics of process scaling to hold. But as the benefits of multicore scaling begin to ebb, a new driver of transistor utility must be found, or the economics of process scaling will break and Moore's Law will end well before we hit final manufacturing limits. An essential question is how much more performance can be extracted from the multicore path in the near future.

This chapter combined technology scaling models, performance models, and empirical results from parallel workloads to answer that question and estimate the remaining performance available from multicore scaling. Using PARSEC benchmarks and ITRS scaling projections, this study predicts best-case average speedup of 7.9 times in a decade at 8 nm. That result translates into a 23% annual performance gain, *for highly parallel workloads* and assuming that each benchmark has its ideal number and granularity of cores.

However, we believe that the ITRS projections are much too optimistic, especially in the challenging sub-22 nanometer environment. The conservative model we use in this chapter more closely tracks recent history. Applying these conservative scaling projections, half of that ideal gain vanishes; the path to 8 nm in 2018 results in a best-case average 3.7× speedup; approximately

14% per year *for highly parallel codes* and *optimal per-benchmark configurations*. The returns will certainly be lower in practice.

Currently, the broader computing community is in consensus that we are in "the multicore era." Consensus is often dangerous, however. Given the low performance returns, adding more cores will not provide sufficient benefit to justify continued process scaling. If multicore scaling ceases to be the primary driver of performance gains at 16 nm (in 2014) the "multicore era" will have lasted a mere ten years, a short-lived attempt to defeat the inexorable consequences of Dennard scaling's failure.

Clearly, architectures that move well past the Pareto-optimal frontier of energy/performance of today's designs will be necessary. Given the time-frame of this problem and its scale, radical departures from conventional approaches are necessary to provide performance an efficiency gains across a wide range of application. There is an emerging synergy between the applications that can tolerate approximation and the unreliability in the computation fabric as technology scales down. If done in a disciplined manner, relaxing the high tax of providing perfect accuracy at the device, circuit, and architecture level can provide a huge opportunity to improve performance and energy efficiency for the domains in which applications can tolerate approximate computation yet deliver acceptable outputs. In this dissertation, we propose techniques that allows general-purpose processors to trade accuracy for gain in both performance and efficiency without disruptive changes to the programming models.

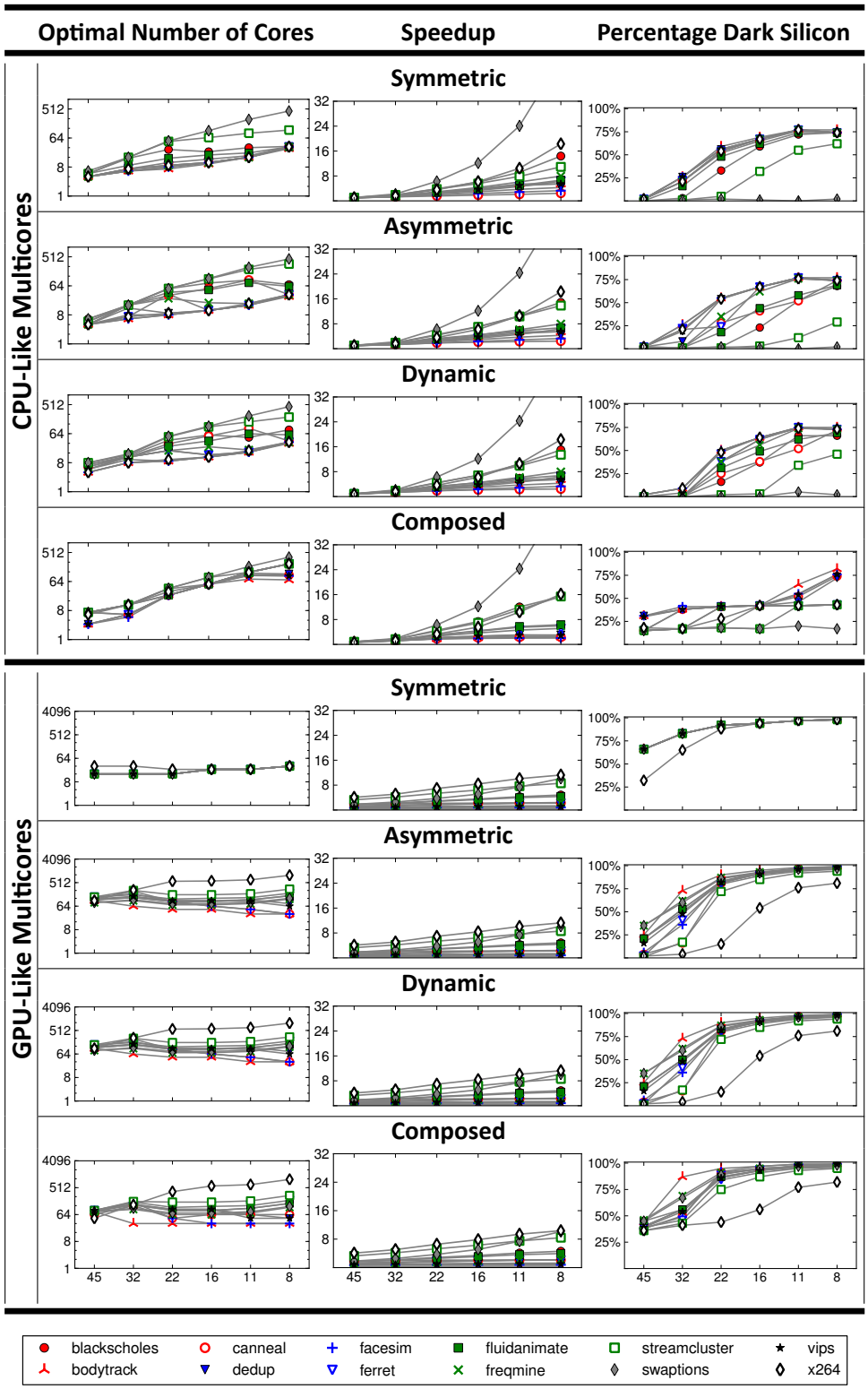


Figure 3.17: Optimal number of cores, speedup over quad-Nehalem at 45 nm, and percentage dark silicon under ITRS scaling projections.

Chapter 4

Variable-Precision von Neumann Architectures

In this chapter, we introduce a variable-precision von Neumann ISA that allows conventional microarchitectures to run many interleavings of approximate and precise instructions. This ISA allows the compiler to convey what can be approximated without specifying how. This abstraction allows the microarchitecture to freely choose from a range of approximation techniques without exposing them to the compiler or the programmer. With this ISA, the compiler segregates approximate operations and storage from the default operations and storage, which adhere to the traditional semantics. The ISA also allows the precise data to be used approximately and vice versa while confining approximation to predictable areas—e.g., address computation of memory operations must always be precise while the data can either be approximate or precise. To avoid the overheads of dynamic correctness checks and error recovery, the ISA is targeted for a disciplined style of approximate programming. Disciplined approximate programming allows programmers to declare which parts of a program can be computed approximately. The compiler provides static guarantees that all approximate computation is properly isolated from the precise computation. That is, the ISA relegates concerns of safety and programmability to the language and compiler. An example of such languages is EnerJ [98]. This chapter also proposes the Truffle microarchitecture that efficiently supports the variable-precision ISA. Our microarchitecture proposal relies on a dual voltage supply for SRAM arrays and logic: a high V_{dd} (leading to accurate operation) and a low V_{dd} (leading to approximate but lower-power operation). We propose a novel dual-voltage SRAM structure and design the microarchitectural extensions to allow every instruction to control its level of precision, i.e. the level of voltage for operations and storage. This chapter provides a framework for conventional von Neumann processors to trade accuracy for energy at the granularity of single instructions.

This chapter is based on work presented in ASPLOS (2012) [40] and is a result of collaboration with Adrian Sampson[a], Luis Ceze[a], and Doug Burger[b].

---

[a] University of Washington

[b] Microsoft Research

4.1 Introduction

Reducing power consumption is becoming a requirement due to limits of device scaling in what is termed the *dark silicon* problem [12, 32]. Furthermore, energy consumption is a first-class concern in computer systems design. Potential benefits go beyond reduced power demands in servers and longer battery life in mobile devices.

Prior work has made significant progress in various aspects of energy efficiency. Hardware optimizations include power gating, voltage and frequency scaling, and sub-threshold operation with error correction [30]. Software efforts have explored managing energy as an explicit resource [118], shutting down unnecessary hardware, and energy-aware compiler optimizations [116]. Raising energy concerns to the programming model can enable a new space of energy savings opportunities.

Trading off quality of service is one technique for reducing energy usage. Allowing computation to be approximate can lead to significant energy savings because it alleviates the "correctness tax" imposed by the wide safety margins on typical designs. Indeed, prior work has investigated hardware and algorithmic techniques for approximate execution [69, 60, 76, 1]. Most applications amenable to energy–accuracy trade-offs (e.g., vision, machine learning, big data analytics, games, etc.) have approximate components, where energy savings are possible, and precise components, whose correctness is critical for application invariants [98, 80].

Recent work has explored language-level techniques to assist programmers in identifying *soft slices*, the parts of programs that may be safely subjected to approximate computation [98]. The hardware is free to use approximate storage and computation for the soft slices without performing dynamic safety checks. Broadly, we advocate co-designing hardware support for approximation with an associated programming model to enable new energy-efficiency improvements while preserving programmability.

In this chapter, we explore how to map disciplined approximate programming models down to an approximation-aware von Neumann microarchitecture. Our architecture proposal includes an ISA extension, which allows a compiler to convey what can be approximated, along with microarchitectural extensions to typical in-order and out-of-order processors that implement the ISA. Our microarchitecture proposal relies on a dual voltage supply for SRAM arrays and logic: a high V_{dd} (leading to accurate operation) and a low V_{dd} (leading to approximate but lower-power oper-

ation). We discuss the implementation of the core structures using dual-voltage primitives as well as dynamic, instruction-based control of the voltage supply. We evaluate the energy consumption and quality-of-service degradation of our proposal using a variety of benchmarks including a game engine, a raytracer, and scientific algorithms.

4.2 An ISA for Disciplined Approximate Computation

With disciplined approximate programming, a program is decomposed into two components: one that runs *precisely*, with the typical semantics of conventional computers, and another that runs *approximately*, carrying no guarantees but only an expectation of best-effort computation. Many applications, such as media processing and machine learning algorithms, can operate reliably even when errors can occur in portions of them [98, 78, 24, 114, 25]. Floating-point data, for example, is by nature imprecise, so many FP-heavy applications have inherent tolerance to error. An architecture supporting disciplined approximation can take advantage of relaxed precision requirements to expose errors that would otherwise need to be prevented or corrected at the cost of energy. By specifying the portion of the application that is tolerant to error, the programmer gives the architecture permission to expose faults when running that part of the program. We propose an ISA that enables a compiler to communicate where approximation is allowed.

Our ISA design is defined by two guiding principles: the ISA should provide an *abstract* notion of approximation by replacing guarantees with informal expectations; and the ISA may be *unsafe*, blindly trusting the programming language and compiler to enforce safety invariants statically.

4.2.1 Replacing Guarantees with Expectations

ISAs normally provide formal *guarantees* for operations (e.g., an "add" instruction must produce the sum of its operands). Approximate operations, however, only carry an *expectation* that a certain operation will be carried out correctly; the result is left formally undefined. For example, an approximate "add" instruction might leave the contents of the output register formally undefined but specify an *expectation* that the result will approximate the sum of the operands. The compiler and programmer may not rely on any particular pattern or method of approximation. Informally,

however, they may expect the approximate addition to be useful in "soft" computation that requires summation.

The lack of strict guarantees for approximate computation is essential for *abstract* approximation. Instructions do not specify which particular energy-saving techniques are used; they only specify where approximation may be applied. Consequently, a fully-precise computer is a valid implementation of an approximation-aware ISA. Compilers for such ISAs can, without modification, take advantage of new approximation techniques as they are implemented. By providing no guarantees for approximate computation, the ISA permits a full range of approximation techniques.

By leaving the kind and degree of approximation unspecified, an approximation-aware ISA could pose challenges for portability: different implementations of the ISA can provide different error distributions for approximate operations. To address this issue, implementations can allow software control of implementation parameters such as voltage (see Section 4.3.3). Profiling and tuning mechanisms could then discover optimal settings for these hardware parameters at application deployment time. This tuning would allow a single application to run at the same level of quality across widely varying approximation implementations.

4.2.2 Responsibilities of the Language and Compiler

An architecture supporting approximate computing requires collaboration from the rest of the system stack. Namely, the architecture relegates concerns of *safety and programmability* to the language and compiler. In order to be usable, approximate computation must be exposed to the programmer in a way that reduces the chance of catastrophic failures and other unexpected consequences. These concerns, while important, can be relegated to the compiler, programming language, and software-engineering tools.

EnerJ [98] is a programming language supporting disciplined approximation. Using a type system, EnerJ provides a static *non-interference* guarantee, ensuring that the approximate part of a program cannot affect the precise portion. In effect, EnerJ enforces separation between the error-tolerant and error-sensitive parts of a program, identifying and isolating the parts that may be subject to relaxed execution semantics. This strict separation brings safety and predictability to programming with approximation. Because it is static, the non-interference guarantee requires no

runtime checking, freeing the ISA (and its implementation) from any need for safety checks that would themselves impose overheads in performance, energy, and design complexity.

In this chapter, we assume that a language like EnerJ is used to provide safety guarantees *statically* to the programmer. The ISA must only expose approximation as an option to the compiler: it does not provide dynamic invariant checks, error recovery, or any other support for programmability. The ISA is thus unsafe *per se*. If used incorrectly, the ISA can produce unexpected results. It is tightly coupled with the compiler and trusts generated code to observe certain invariants. This dependence on static enforcement is essential to the design of a simple microarchitecture that does not waste energy in providing approximation.

4.2.3 Requirements for the ISA

An ISA extension for disciplined approximate programming consists of new instruction variants that leave certain aspects of their behavior undefined. These new instructions must strike a balance between energy savings and usability: they must create optimization opportunities through strategic use of undefined behavior but not be so undefined that their results could be catastrophic. Namely, approximate instructions should leave certain data values undefined but maintain predictable control flow, exception handling, and other bookkeeping.

To support a usable programming model similar to EnerJ [98], an approximation-aware ISA should exhibit the following properties:

- Approximate computation must be controllable at an *instruction granularity*. Approximation is most useful when it can be interleaved with precise computation. For example, a loop variable increment should likely be precise while an arithmetic operation in the body of the loop may be approximate. For this reason, it must be possible to mark individual instructions as either approximate or precise.

- The ISA must support *approximate storage*. The compiler should be able to instruct the ISA to store data approximately or precisely in registers, caches, and main memory.

- It must be possible to *transition* data between approximate and precise storage. (In EnerJ, approximate-to-precise movement is only permitted using an explicit programmer “endorse-

Table 4.1: ISA extensions for disciplined approximate programming. These instructions are based on the Alpha instruction set.

Group	Approximate Instruction
Integer load/store	LDx.a, STx.a
Integer arithmetic	ADD.a, CMPEQ.a, CMPLT.a, CMPLE.a, MUL.a, SUB.a
Logical and shift	AND.a, NAND.a, OR.a, XNOR.a NOR.a, XOR, CMOV.a, SLL.a, SRA.a, SRL.a
Floating point load/store	LDF.a, STF.a
Floating point operation	ADDF.a, CMPF.x, DIVF.a, MULF.a, SQRTF.a, SUBF.a, MOV.a, CMOV.a, MOVFI.a, MOVIF.a

ment," but this annotation has no runtime effect.) For full flexibility, the ISA must permit programs to use precise data approximately and vice-versa.

- Precise instructions, where approximation is not explicitly enabled, must carry traditional semantic guarantees. The effects of approximation must be constrained to where it is requested by the compiler.

- Approximation must be confined to predictable areas. For example, address computation for memory accesses must always be precise; approximate store instructions should not be allowed to write to arbitrary memory locations. Approximate instructions must not carry semantics so relaxed that they cannot be used.

4.2.4 ISA Extensions for Approximation

Table 4.1 summarizes the approximate instructions that we propose adding to a conventional architecture. Without loss of generality, we assume an underlying Alpha ISA [21].

Approximate operations. The extended ISA provides approximate versions of all integer arithmetic, floating-point arithmetic, and bitwise operation instructions provided by the original ISA. These instructions have the same form as their precise equivalents but carry no guarantees about their output values. The approximate instructions instead carry the informal expectation of approximate adherence to the original instructions' behavior. For example, ADD.a takes two arguments and produces one output, but the ISA makes no promises about what that output will be. The instruction may be expected to typically perform addition but the programmer and compiler may

not rely on any consistent output behavior. In practice, we expect each approximate instructions exhibit error levels that can be measured with probability distributions. As long as the probability, magnitude, and frequency of errors are low enough, the programmer can still expect acceptable results. However, no formal guarantees are provided.

Approximate registers. Each register in the architecture is, at any point in time, in either *precise mode* or *approximate mode*. When a register is in approximate mode, reads are not guaranteed to obtain the exact value last written, but there is an expectation that the value is likely the same.

The compiler does not explicitly set register modes. Instead, the precision of a register is implicitly defined based on the precision of the last instruction that wrote to it. In other words, a precise operation makes its destination register precise while an approximate operation puts its destination register into approximate mode.

While register precision modes are set implicitly, the precision of operand accesses must be declared explicitly. Every instruction that takes register operands is extended to include an extra bit per operand specifying the operand's precision. This makes precision level information available to the microarchitecture *a priori*, drastically simplifying the implementation of dual-voltage registers (see Section 4.4.1). It does not place a significant burden on the compiler as the compiler must statically determine registers' precision anyway. If the correspondence between register modes and operand accesses is violated, the value is undefined (see below).

With this design, data can move freely between approximate and precise registers. For example, a precise ADD instruction may use an approximate register as an operand; this transition corresponds to an *endorsement* in the EnerJ language. The opposite transition, in which precise data is used in approximate computation, is also permitted and frequently occurs in EnerJ programs.

Approximate loads, stores, and caching. The ISA defines a *granularity of approximation* at which the architecture supports setting the precision of cached memory. In practice, this granularity will likely correspond to the smallest cache line size in the processor's cache hierarchy.[1] For example, if an architecture has 16-byte L1 cache lines and supports varying the precision of every cache line,

[1] Note that architects generally avoid tying cache line size to the ISA. However, we believe that in cases of strong co-design between architecture and compiler such as ours, it is acceptable to do so.

then it defines the approximation granularity to be 16 bytes. Each region of memory aligned to the granularity of approximation (hereafter called an *approximation line* for brevity) is in either approximate or precise mode at any given time.

An approximation line's precision mode is implicitly controlled by the precision of the loads and stores that access it. In particular, the ISA guarantees reliable data storage for precise accesses if, for every load from line x, the preceding store to line x has the same precision. (That is, after a precise store to x, only precise loads may be issued to x until the next store to x.) For the compiler and memory allocator, this amounts to ensuring that precise and approximate data never occupy the same line. Memory allocation and object layout must be adapted to group approximate data to line-aligned boundaries. Statically determining each line's precision is trivial for any language with sufficiently strong static properties. In EnerJ specifically, a type soundness theorem implies that the precision of every variable is known statically for every access.

The ISA requires this pattern of consistent accesses in order to simplify the implementation of approximation-aware caching. Specifically, it allows the following simple cache-approximation policy: a line's precision is set by misses and writes but is not affected by read hits. During read hits, the cache can assume that the precision of the line matches the precision of the access.

Approximate loads and stores may read and write arbitrary values to memory. Accordingly, precise stores always write data reliably, but a precise load only reads data reliably when it accesses a line in precise mode. However, any store (approximate or precise) can only affect the address it is given: address calculation and indexing are never approximated.

Approximate main memory. While this chapter focuses on approximation in the core, main memory (DRAM modules) may also support approximate storage. The refresh rate of DRAM cells, for example, may be reduced so that data is no longer stored reliably [80]. However, memory approximation is entirely decoupled from the above notion of approximate caching and load/store precision. This way, an approximation-aware processor can be used even with fully-precise memory. Furthermore, memory modules are likely to support a different granularity for approximation from caches—DRAM lines, for instance, typically range from hundreds of bytes to several kilobytes. Keeping memory approximation distinct from cache approximation decouples the memory from specific architecture parameters.

The program controls main-memory approximation *explicitly*, using either a special instruction or a system call to manipulate the precision of memory regions. Loads and stores are oblivious to the memory's precision; the compiler is responsible for enforcing a correspondence. When main-memory approximation is available, the operating system may store precision levels in the page table to preserve the settings across page evictions and page faults.

Preservation of precision. In several cases, where the ISA supports both precise and approximate operation, it relies on the compiler to treat certain data consistently as one or the other. For example, when a register is in approximate mode, all instructions that use that register as an operand must mark that operand as approximate. Relying on this correspondence simplifies the implementation of approximate SRAM arrays (Section 4.4.1).

The ISA does not enforce precision correspondence. No exception is raised if it is violated. Instead, as with many other situations in the ISA, the resulting value from any inconsistent operation is left undefined. Unlike other approximate operations, however, the *informal expectation* in these situations is also weaker: precise reads from approximate-mode registers, for example, should be expected to return random data. The compiler should avoid these situations even when performing approximate computation.

These situations constitute violations of precision correspondence:

- A register in approximate mode is used as a precise operand to an instruction. (Note that a precise instruction can use an approximate operand; the operand must then be declared as approximate.)

- Conversely, a precise-mode register is used as an approximate operand.

- An approximate load (e.g., LDW.a) is issued to an address in an approximation line that is in precise mode.

- Conversely, a precise load is issued to an approximate-mode line.

In every case, these are situations where undefined behavior is already present due to approximation, so the ISA's formal guarantees are not affected by this choice. Consistency of precision only

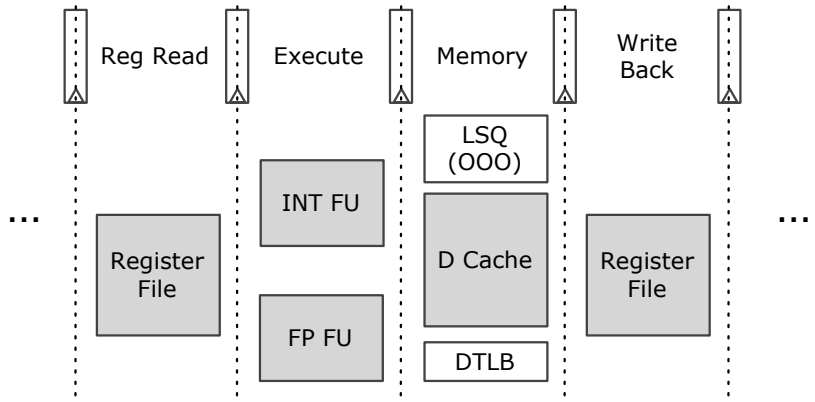


Figure 4.1: The data movement/processing plane of the processor pipeline. Approximation-aware structures are shaded. The instruction control plane stages (Fetch and Decode, as well as Rename, Issue, Schedule, and Commit in the OOO pipeline) are not shown.

constitutes a recommendation to the compiler that these situations be avoided. These weak semantics keep the microarchitecture simple by alleviating the need for precision state checks (see Section 4.4.1).

4.3 Design Space

As discussed above, formally, approximate instructions may produce arbitrary results. For example, ADD.a may place any value in its destination register. However, approximate instructions still have defined semantics: ADD.a cannot modify any register other than its output register; it cannot raise a divide-by-zero exception; it cannot jump to an arbitrary address. These guarantees are necessary to make our approximation-aware ISA usable.

Our microarchitecture must carefully distinguish between structures that can have relaxed correctness and those for which reliable operation is always required. Specifically, all fetched instructions need to be decoded precisely and their target and source register indices need to be identified without error. However, the content of those registers may be approximate and the functional units operating on the data may operate approximately. Similarly, memory addresses must be error-free, but the data retrieved from the memory system can be incorrect when that data is marked as approximate. Consequently, we divide the microarchitecture into two distinct planes: the *instruction control plane* and the *data movement/processing plane*. As depicted in Figure 4.1, the data movement/processing plane comprises the register file, data caches, load store queue, functional units,

and bypass network. The instruction control plane comprises the units that fetch, decode, and perform necessary bookkeeping for in-flight instructions. The instruction control plane is kept precise, while the data movement/processing plane can be approximate for approximate instructions. Since the data plane needs to behave precisely or approximately depending on the instruction being carried out, the microarchitecture needs to do some bookkeeping to determine when a structure can behave approximately.

This chapter explores voltage reduction as a technique for saving energy. (Other techniques, such as aggressive timing closure or reducing data width, are orthogonal.) Each frequency level (f_{max}) is associated with a minimum voltage (V_{min}) and lowering the voltage beyond that may cause timing errors. Lowering the voltage reduces energy consumption quadratically when the frequency is kept constant. However, we cannot lower the voltage of the whole processor as this would cause errors in structures that need to behave precisely. This leads to the core question of how to provide precise behavior in the microarchitecture.

One way to provide precise behavior, which we explore in this chapter, is to run critical structures at a safe voltage. Alternatively, error correction mechanisms could be used for critical structures and disabled for the data movement/processing plane while executing approximate instructions. This way, the penalty of error checking would not be incurred for approximate operations. However, if V_{dd} were low enough to make approximation pay off, many expensive recoveries would be required during precise operations. For this reason, we propose using two voltage lines: one for precise operation and one for approximate operation. Below we describe two alternative designs for a dual-voltage microarchitecture.

4.3.1 Unchecked Dual-Voltage Design

Our dual-voltage microarchitecture, called Truffle, needs to guarantee that (1) the instruction control remains precise at all times, and (2) the data processing plane structures lower precision only when processing approximate instructions. Truffle has two voltages: a nominal, reliable level, referred to as $V_{dd}H$, and a lower level, called $V_{dd}L$, which may lead to timing errors. All structures in the instruction control plane are supplied $V_{dd}H$, and, depending on the instruction being executed, the structures in the data processing plane are dynamically supplied $V_{dd}H$ or $V_{dd}L$. The detailed

design of a dual-voltage data processing plane, including the register file, data cache, functional units, and bypass network, is discussed in the next section.

4.3.2 Checked Dual Voltage Design

The energy savings potential of Truffle is limited by the proportion of power used by structures that operate precisely at $V_{dd}H$. Therefore, reducing the energy consumption of precise operations will lead to higher relative impact of using approximation. This leads to another possible design point, which lowers the voltage of the instruction control plane beyond $V_{dd}H$ but not as aggressively as $V_{dd}L$ and employs error correction to guarantee correct behavior using an approach similar to Razor [30]. We refer to this conservative level of voltage as $V_{dd}L_{high}$. The data plane also operates at $V_{dd}L_{high}$ when running the precise instructions and $V_{dd}L$ when running the approximate instructions. Since the instruction control plane operates at the reliable voltage level, the components in this plane need to be checked and corrected in the case of any errors. The same checking applies to the data movement/processing plane while running precise instructions. While this is an interesting design point, we focus on the unchecked dual-voltage design due to its lower complexity.

4.3.3 Selecting $V_{dd}L$

In the simplest case, $V_{dd}L$ can be set statically at chip manufacture and test time. However, the accuracy of approximate operations depends directly on this value. Therefore, a natural option is to allow $V_{dd}L$ to be set dynamically depending on the QoS expectations of the application. Fine-grained voltage adjustments can be performed after fabrication using off-chip voltage regulators as in the Intel SCC [63] or by on-chip voltage regulators as proposed by Kim et al. [72]. Off-chip voltage regulators have a high latency while on-chip voltage regulators provide lower-latency, fine-grained voltage scaling. Depending on the latency requirement of the application and the type of regulator available, $V_{dd}L$ can be selected at deployment time or during execution. Future work should explore software-engineering tools that assist in selecting per-application voltage levels.

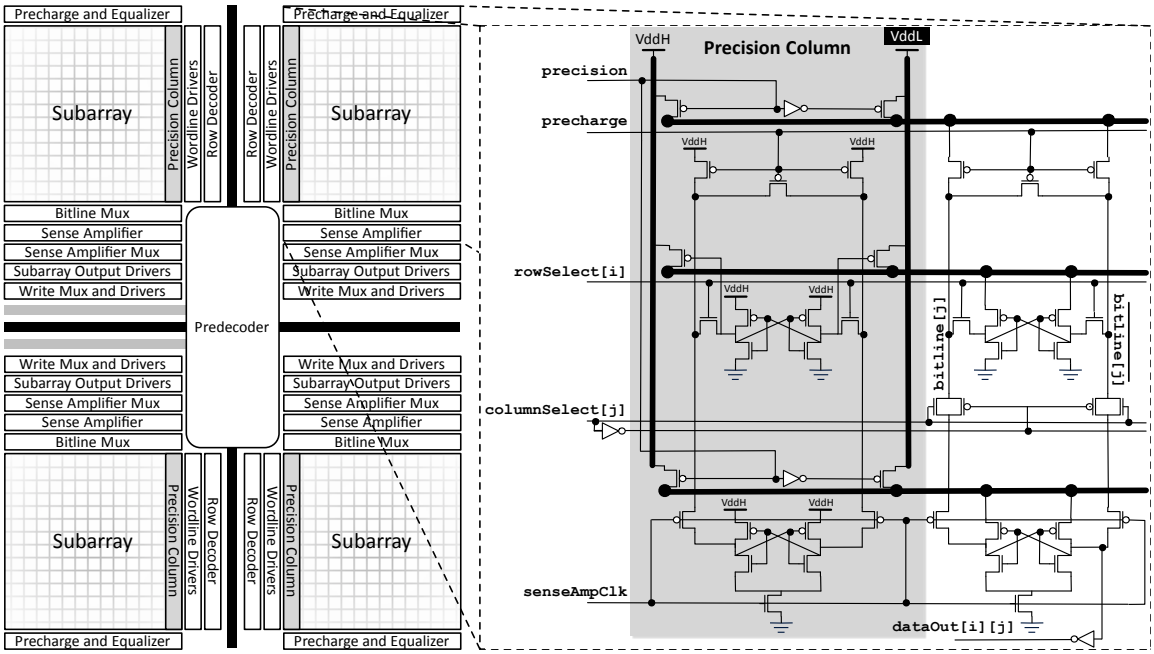


Figure 4.2: Dual-voltage mat, consisting of four identical dual-voltage subarrays, and partial transistor-level design of the subarrays and the *precision column*, which is shaded. The power lines during a read access are shown in bold.

4.4 Truffle: A Dual-Voltage Microarchitecture for Disciplined Approximation

This section describes Truffle in detail. We start with the design of a dual-voltage SRAM structure, which is an essential building block for microarchitectural components such as register files and data caches. Next, we discuss the design of dual-voltage multiplexers and level shifters. We then address the design of structures in both in-order and OOO Truffle cores, emphasizing how the voltage is selected dynamically depending on the precision level of each instruction. Finally, we catalog the microarchitectural overheads imposed by Truffle's dual-voltage design.

4.4.1 Dual-Voltage SRAM Array

We propose dual-voltage SRAM arrays, or DV-SRAMs, which can hold precise and approximate data simultaneously. Like its single-voltage counterpart, a DV-SRAM array is composed of mats. A mat, as depicted in Figure 4.2, is a self-contained memory structure composed of four identical subarrays and associated predecoding logic that is shared among the subarrays. The data-in/-out and address

lines typically enter the mat in the middle. The predecoded signals are forked off from the middle to the subarrays. Figure 4.2 also illustrates a circuit diagram for a single subarray, which is a two-dimensional array of single-bit SRAM cells arranged in rows and columns. The highlighted signals correspond to a read access, which determines the critical path of the array.

For any read or write to a DV-SRAM array, the address goes through the predecoding logic and produces the one-hot **`rowSelect`** signals along with the **`columnSelect`** signals for the column multiplexers. During read accesses, when a row is activated by its **`rowSelect`** signal, the value stored in each selected SRAM cell is transferred over two complementary **`bitline`** wires to the sense amplifiers. Meanwhile, the **`bitlines`** have been precharged to a certain V_{dd}. Each sense amplifier senses the resulting swing on the **`bitlines`** and generates the subarray output. The inputs and outputs of the sense amplifiers may be multiplexed depending on the array layout. The sense amplifiers drive the **`dataOut`** of the mat for a read access, while **`dataIn`** drives the **`bitlines`** during a write access.

To be able to store both approximate and precise data, we divide the data array logic into two parts: the indexing logic and the data storage/retrieval logic. To avoid potential corruption of precise data, the indexing logic *always* needs to be precise, even when manipulating approximate data. The indexing logic includes the address lines to the mats, the predecoding/decoding logic (row and column decoders), and the **`rowSelect`** and **`columnSelect`** drivers. The data storage/retrieval logic, in contrast, needs to alternate between precise and approximate mode. Data storage/retrieval includes the precharge/equalizing logic, SRAM cells, bitline multiplexers, sense amplifiers, **`dataOut`** drivers and multiplexers, and **`dataIn`** drivers. For approximate accesses, this set of logic blocks operates at $V_{dd}L$.

In each subarray of a mat, a row of bits (SRAM cells) is either at high voltage ($V_{dd}H$) or low voltage ($V_{dd}L$). The precision column in Figure 4.2, which is a single-bit column operating at $V_{dd}H$, stores the voltage state of each row. The precision column is composed of 8-transistor cells: 6-transistor SRAM cells each augmented by two PMOS transistors. In each row, the output of the precision column drives the power lines of the entire row. This makes it possible to *route only one power line* to the data rows as well as the prechargers and sense amplifiers. This way, the extra $V_{dd}L$ power line is only routed to the precision column in each subarray and is distributed to the

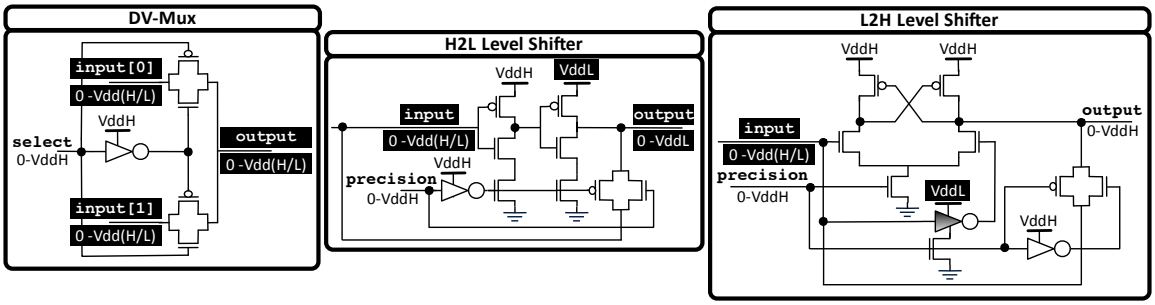


Figure 4.3: Transistor-level design of dual-voltage multiplexer (DV-Mux) and high-to-low (H2L) and low-to-high (L2H) level shifters.

rows through the precision cells, which significantly reduces the overhead of running two voltage lines.

For a read access, the **bitlines** need to be precharged to the same voltage level as the row being accessed. Similarly, the sense amplifiers need to operate at the same voltage level as the subarray row being accessed. A **precision** signal is added to the address bits and routed to all the mats. The **precision** signal presets the voltage levels of the precharge and equalizing logic and the sense amplifiers before the address is decoded and the **rowSelect** and **columnSelect** signals are generated. This voltage presetting ensures that the sense amplifiers are ready when the selected row puts its values on the **bitlines** during a read. Furthermore, during a write, the value in the precision column is set based on the **precision** signal.

Since the precision cell is selected at the same time as the data cells, the state stored in the precision column cannot be used to set the appropriate voltage levels in the sense amplifiers and prechargers. Therefore, the precision information needs to be determined *before* the read access starts in the subarrays. This is why our ISA extensions (Section 4.2) include a precision flag for each source operand: these flags, along with the precision levels of instructions themselves, are used to set the **precision** signal when accessing registers and caches.

4.4.2 Voltage Level Shifting and Multiplexing

Several structures in the Truffle microarchitecture must be able to deal with both precise and approximate data. For this reason, our design includes *dual-voltage multiplexers*, which can multiplex signals of differing voltage, and *voltage level shifters*, which enable data movement between the high- and low-voltage domains.

Figure 4.3 illustrates the transistor-level design of the single-bit, one-input dual-voltage multiplexers (DV-Mux) as well as the high-to-low (H2L) and low-to-high (L2H) level shifters. The select line of the multiplexer and its associated inverter operate at $V_{dd}H$, while the input data lines can swing from 0 V to either $V_{dd}L$ or $V_{dd}H$. The L2H level shifter is a conventional differential low-to-high level shifter and the H2L level shifter is constructed from two back-to-back inverters, one operating at $V_{dd}H$ and the other operating at $V_{dd}L$. In addition to the **`input`** signal, the level shifters take an extra input, **`precision`**, to identify the voltage level of **`input`** and disengage the level-shifting logic to prevent unnecessary power consumption. For example, in the L2H level shifter, when **`precision`** is 0, **`input`** is precise (0 V or $V_{dd}H$) and does not require any level shifting, (**`output`** $\leftarrow$ **`input`**). However, when **`precision`** is 1, the level shifter is engaged and generates the **`output`** with the required voltage level.

4.4.3 Truffle's Microarchitectural Constructs

This section describes the Truffle pipeline stage-by-stage, considering both in-order and out-of-order implementations. The out-of-order design is based on the Alpha 21264 [70] and uses a tag-and-index register renaming approach. We highlight whether a pipeline stage belongs to the instruction control plane or the data movement/processing plane. Importantly, we discuss how the voltage is dynamically selected in the structures that support approximation.

Fetch (OOO/in-order) [Instruction Control Plane]. The fetch stage belongs to the instruction control plane and is identical to a regular OOO/in-order fetch stage. All the components of this stage, including the branch predictor, instruction cache, and ITLB, are ordinary, single-voltage structures. Approximate instructions are fetched exactly the same way as precise instructions.

Decode (OOO/in-order) [Instruction Control Plane]. The instruction decoding logic needs to distinguish between approximate and precise instructions. The decode stage passes along one extra bit indicating the precision level of the decoded instruction.

In addition, based on the instruction, the decoder generates precision bits to accompany the indices for each register read or written. These register precision bits will be used when accessing the dual-voltage register file as discussed in Section 4.4.1. The precision levels of the source

registers are extracted from the operand precision flags while the precision of the destination register corresponds to the precision of the instruction. For load and store instructions, the address computation must always be performed precisely, even when the data being loaded or stored is approximate. For approximate load and store instructions, the registers used for address calculation are always precise while the data register is always approximate. Recall that the microarchitecture does not check that precise registers are always used precisely and approximate registers are used approximately. This correspondence is enforced by the compiler and encoded in the instructions, simplifying the design and reducing the overheads of mixed-precision computation.

Rename (OOO) [Instruction Control Plane]. For the OOO design, in which the register renaming logic generates the physical register indices/tags, the physical register tags are coupled with the register precision bits passed from the decode stage. The rest of the register renaming logic is the same as in the base design.

Issue (OOO) [Instruction Control Plane]. The slots in the issue logic need to store the register index precision bits as well as the physical register indices. That is, each physical tag entry in an issue slot is extended by one bit. The issue slots also need to store the bit indicating the precision of the instruction. When an approximate load instruction is issued, it gets a slot in the load/store queue. The address entry of each slot in the load/store queue is extended by one extra bit indicating whether the access is approximate or precise. The precision bit is coupled with the address in order to control the precision level of the data cache access (as described above in Section 4.4.1).

Schedule (OOO) [Instruction Control Plane]. As will be discussed in the execution stage, there are separate functional units for approximate and precise computation in Truffle. The approximate functional units act as *shadows* of their precise counterparts. The issue width of the processor is *not* extended due to the extra approximate functional units and no complexity is added to the scheduling logic to accommodate them. The coupled approximate/precise functional units appear to the scheduler as a single unit. For example, if an approximate floating-point operation is scheduled to issue, the precise and approximate FPUs are both considered busy. The bit indicating the

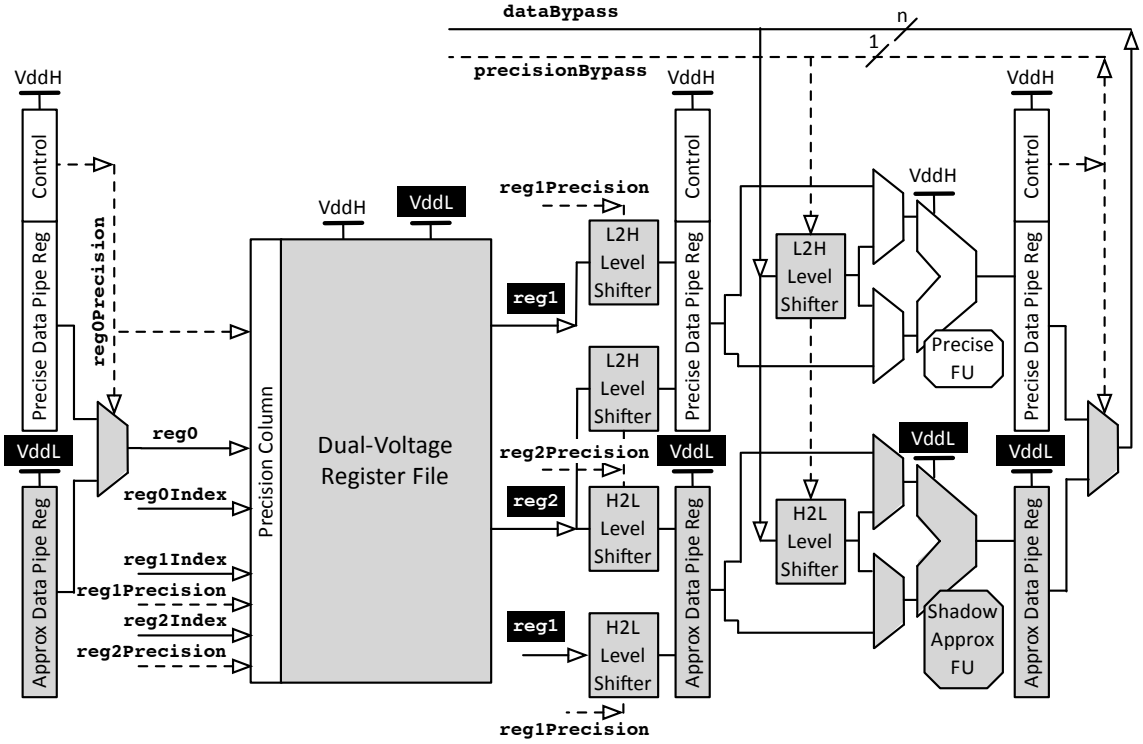


Figure 4.4: The register and execution stages in the Truffle pipeline along with the bypass network. The DV-Muxes and other approximation-aware units are shaded. The single-bit precision signals in the bypass network and in each stage are dashed lines.

precision level of the instruction is used to enable either the approximate or precise functional unit exclusively.

Register Read (OOO/in-order) [Data Movement/Processing Plane]. The data movement/processing plane starts at the register read stage. As depicted in Figure 4.4, the pipeline registers are divided into two sets starting at this stage: one operating at $V_{dd}L$ (Approx Data Pipe Reg) and the other operating at $V_{dd}H$ (Precise Data Pipe Reg + Control). The approximate pipeline register holds approximate data. The precise pipeline register contains control information passed along from previous stages and precise program data. The outputs of the precise and approximate pipeline registers are multiplexed through a DV-Mux as needed.

The dual-voltage register file (physical in the OOO design and architectural in the in-order design) is made up of DV-SRAM arrays. Each register can be dynamically set as approximate or precise. While the precision levels of the general-purpose registers are determined dynamically, the

pipeline registers are hardwired to their respective voltages to avoid voltage level changes when running approximate and precise instructions back-to-back.

The ISA allows precise instructions to use approximate registers as operands and vice-versa. To carry out such instructions with minimal error, the voltage levels of the operands need to be changed to match the voltage level of the operation. In the register read stage, the level shifters illustrated in Figure 4.4 convert the voltage level of values read from the register file. For example, if **`reg1`** is precise and used in an approximate operation, its voltage level is shifted from high to low through an H2L level shifter before being written to an approximate pipeline register. The precision level of **`reg1`**, denoted by **`reg1Precision`**, is passed to the level shifter to avoid unnecessary level shifting in the case that **`reg1`** is already approximate. Similarly, a low-to-high (L2H) level shifter is used to adjust the voltage level of values written to the precise pipeline register. Note that only *one* of the approximate or precise data pipeline registers is enabled based on the precision of the instruction.

Execute (OOO/in-order) [Data Movement/Processing Plane]. As shown in Figure 4.4, all the functional units are duplicated in the execution stage. Half of them are hardwired to $V_{dd}H$, while the other half operate at $V_{dd}L$ as *shadows*. That is, the scheduler does not distinguish between a shadow approximate FU and its precise counterpart. Only the instruction precision bit controls whether the results are taken from the precise or approximate FU. Low-voltage functional units are connected to a low-voltage pipeline register. The outputs of the approximate FUs connect to the *same* broadcast network as their precise counterparts through a dual-voltage multiplexer driven by the approximate and precise pipeline register pair at the end of the execution stage.

The inputs of functional units may also be driven by the bypass network. Level shifters at the inputs of the functional units adjust the level of the broadcasted input data using the broadcasted precision bit. Only a single-bit precision signal is added to the broadcast network. Because the output of each FU pair is multiplexed, the extra FUs do not increase the size of the broadcast network beyond adding this single-bit precision line. While the data bypass network alternates between $V_{dd}L$ and $V_{dd}H$, the tag forwarding network in the OOO design always works at the high voltage level since it carries necessarily-precise register indexing information.

To avoid unnecessary dynamic power consumption in the functional units, the input to one functional unit is kept constant (by not writing to its input pipeline register) when its opposite-precision counterpart is being used.

An alternative design could use dual-voltage functional units and change the voltage depending on the instruction being executed. This would save area and static power but require a more complex scheduler design that can set the appropriate voltage level in the functional unit before delivering the operands to it. Since functional units can be large and draw a considerable amount of current while switching, a voltage-selecting transistor for a dual-voltage functional unit needs to be sized such that it can provide the required drive. Such transistors tend to consume significant power. Another possibility is lowering the FU voltage based on phase prediction. When the application enters an approximate phase, the voltage level of the functional unit is lowered. Phase prediction requires extra power and complicates the microarchitecture design. The main objective in the design of Truffle is to keep the microarchitectural changes to a minimum and avoid complexity in the instruction control and bookkeeping. Furthermore, since programs consist of a mixture of precise and approximate instructions, it is not obvious that phase-based voltage level adjustments can provide benefit that compensates for the phase prediction overhead. Additionally, static partitioning can help tolerate process variation by using defective functional units for approximate computations.

Memory (OOO/in-order) [Data Movement/Processing Plane]. As previously discussed, the address field in each slot of the load/store queue is extended by one bit that indicates whether the address location is precise or approximate. The data array portion of the data cache is a DV-SRAM array, while the tag array is an ordinary single-voltage SRAM structure. The approximation granularity in the data cache is a cache line: one extra bit is stored in the tag array which identifies the precision level of each cache line. The extra bit in the load/store queue is used as the precision signal when accessing the DV-SRAM data array. The miss buffer, fill buffer, prefetch buffer, and write back buffers all operate at $V_{dd}H$. The DTLB also requires no changes.

The precision level of a cache line is determined by the load or store instruction that fills the line. If an approximate access misses in the cache, the line is fetched as approximate. Similarly, a precise miss fills the cache line as precise. Subsequent write hits also affect line precision: when

a store instruction modifies the data in a cache line, it also modifies the line's precision level (see Section 4.2).

This chapter focuses on the Truffle core and does not present a detailed design for approximation-aware lower levels of cache or main memory. The L1 cache described here could work with an unmodified, fully-precise L2 cache, but a number of options are available for the design of an approximation-aware L2. If the L2 has the same line size as the L1, then an identical strategy can be applied. However, L2 lines are often larger than L1 lines. In that case, one option is to control L2 precision at a sub-line granularity: if the L2 line size is n times the L1 line size, then the L2 has n precision columns. An alternative design could pair the lower-level caches with main memory, setting the precision of L2 lines based on the explicitly-controlled main-memory precision levels. These non-core design decisions are orthogonal to the core Truffle design and are an avenue for future work.

Write Back (OOO/in-order) [Data Movement/Processing Plane]. The write back value comes from the approximate or precise pipeline register. As shown in Figure 4.4, the dual-voltage multiplexer driving **`reg0`** forwards the write back value to the data bypass network. The precision bit accompanying the write back value (**`reg0Precision`**) is also forwarded over the bypass network's precision line.

Commit (OOO) [Instruction Control Plane]. The commit stage does not require any special changes and the reorder buffer slots do not need to store any extra information for the approximate instructions. The only consideration is that, during rollbacks due to mispredictions or exceptions, the precision state of the registers (one bit per register) needs to be restored. Another option is to restore all the registers as precise, ignoring the precision level of the registers during rollback.

4.4.4 Microarchitectural Overheads in Truffle

In this section, we briefly consider the microarchitectural overheads imposed by routing two power lines to structures in Truffle's data movement/processing plane. First, the data part of the pipeline registers is duplicated to store approximate data. An extra level of dual-voltage multiplexing is added after these pipeline registers to provide the data with the correct voltage for the next stage.

The DV-SRAM arrays, including the register file and data array in the data cache, store one precision bit per row. In addition, a one-bit precision signal is added to each read/write port of the DV-SRAM array and routed to each subarray along with the $V_{dd}L$ power line. The tag for each data cache line is also augmented by a bit storing the precision state of the line. However, the tag array itself is an ordinary single-voltage SRAM structure. To adjust the voltage level of the operands accessed from the dual-voltage register file, one set of H2L and one set of L2H level shifters is added for each read port of the register file. Similarly, in the execution stage, one set of each level shifter type is added per operand from the data bypass network. The data bypass network is also augmented with a single-bit precision signal per operand. In addition, each entry in the issue queue is extended by one bit preserving the precision level of the instruction. Each physical tag entry in an issue slot is also extended by one precision bit. Similarly, the slots in the load/store queue are augmented by one extra bit indicating whether the access is approximate or precise. The pipeline registers also need to store the precision level of the operands and instructions.

4.5 Experimental Results

Our goals in evaluating Truffle are to determine the energy savings brought by disciplined approximation, characterize where the energy goes, and understand the QoS implications for software.

4.5.1 Evaluation Setup

We extended CACTI [86] to model the dual-voltage SRAM structure we propose and incorporated the resulting models into McPAT [77]. We modeled Truffle at the 65 nm technology node in the context of both in-order and out-of-order (based on the 21264 [70]) designs. Table 4.2 shows the detailed microarchitectural parameters.

Disciplined approximate computation represents a trade-off between energy consumption and application output quality. Therefore, we evaluate each benchmark for two criteria: energy savings and sensitivity to error. For the former, we collect statistics from the benchmarks' execution as parameters to the McPAT-based power model; for the latter, we inject errors into the execution and measure the consequent degradation in output quality.

Table 4.2: Microarchitectural parameters.

Parameter	OOO Truffle	In-order Truffle
Fetch/Decode Width	4/4	2/2
Issue/Commit Width	6/4	—/—
INT ALUs/FPUs	4/2	1/1
INT Mult/Div Units	1	1
Approximate INT ALUs/FPUs	4/2	1/1
Approximate INT Mult/Div Units	1	1
INT/FP Issue Window Size	20/15	—
ROB Entries	80	—
INT/FP Architectural Registers	32/32	32/32
INT/FP Physical Registers	80/72	—
Load/Store Queue Size	32/32	—
ITLB	128	64
I Cache Size	64 Kbyte	16 Kbyte
Line Width/Associativity	32/2	16/4
DTLB	128	64
D Cache Size	64 Kbyte	32 Kbyte
Line Width/Associativity	16/2	16/4
Branch Predictor	Tournament	Tournament

For both tasks, we use a source-to-source transformation that instruments the program for statistics collection and error injection. Statistics collected for power modeling include variable, field, and array accesses, basic blocks (for branches), and arithmetic and logical operators. A cache simulator is used to distinguish cache hits and misses; the register file is simulated as a small, fully-associative cache. For error injection, each potential injection point (approximate operations and approximate memory accesses) is intercepted; each bit in the resulting value is flipped according to a per-component probability before being returned to the program.

Benchmarks. We examine nine benchmark programs written in the EnerJ language, which is an extension to Java [98] (see Table 4.3). The applications are the same programs that were evaluated in [98]: existing Java programs hand-annotated with approximate type qualifiers that distinguish their approximate parts. Five of the benchmarks come from the SciMark2 suite. ZXing is a multi-format bar code recognizer developed for Android smartphones. jMonkeyEngine is a game development engine; we examine the framework's triangle-intersection algorithm used for collision detection. ImageJ is a library and application for raster image manipulation; we examine its flood-filler algorithm. Finally, we examine a simple 3D raytracer.

For each program, an application-specific *quality-of-service metric* is defined in order to quantify the loss in output quality caused by hardware approximation. For most of the applications,

Table 4.3: List of benchmarks.

Application	Description	Type
fft	SciMark2 benchmark: scientific kernels	FP
sor		FP
mc		FP
smm		FP
lu		FP
zxing	Bar code decoder for mobile phones	FP/integer
jmeint	jMonkeyEngine game framework: triangle intersection kernel	FP
imagefill	ImageJ raster image processing application: flood-filling kernel	integer
raytracer	3D image renderer	FP

the metric is the root-mean-square error of the output vector, matrix, or pixel array. For jmeint, the jMonkeyEngine triangle-intersection algorithm, the metric is the proportion of incorrect intersection decisions. Similarly, for the zxing bar code recognizer, it is the proportion of unsuccessful decodings of a sample QR code image.

4.5.2 Unchecked Truffle Microarchitecture

Figure 4.5 presents the energy savings achieved in the core and L1 cache for the unchecked dual-voltage Truffle microarchitecture in both OOO and in-order configurations. In both designs, $V_{dd}H = 1.5$ V and $V_{dd}L$ takes values that are 50%, 62.5%, 75%, and 87.5% of $V_{dd}H$. The frequency is set constant at 1666 MHz. The Truffle cores include DV-SRAM arrays and extra approximate functional units. The baseline for the reported energy savings is the same core operating at the reliable voltage level of 1.5 V and 1666 MHz *without* the extra functional units or dual-voltage register files and data cache. Our model assumes that Truffle's microarchitectural additions do not prolong the critical path of the base design.

Depending on the configuration, voltage, and application, Truffle ranges from increasing energy by 5% to saving 43%. For the in-order configuration, all voltage levels lead to energy savings; the OOO design shows energy savings when $V_{dd}L$ is less than 75% of $V_{dd}H$. Our results suggest Truffle exhibits a "break even" point at which its energy savings outweigh its overheads.

The difference between the energy savings in OOO and in-order Truffle cores stems from the fact that the instruction control plane in the OOO core accounts for a much larger portion of total energy than in the in-order core. Recall that the instruction control plane in the OOO core includes

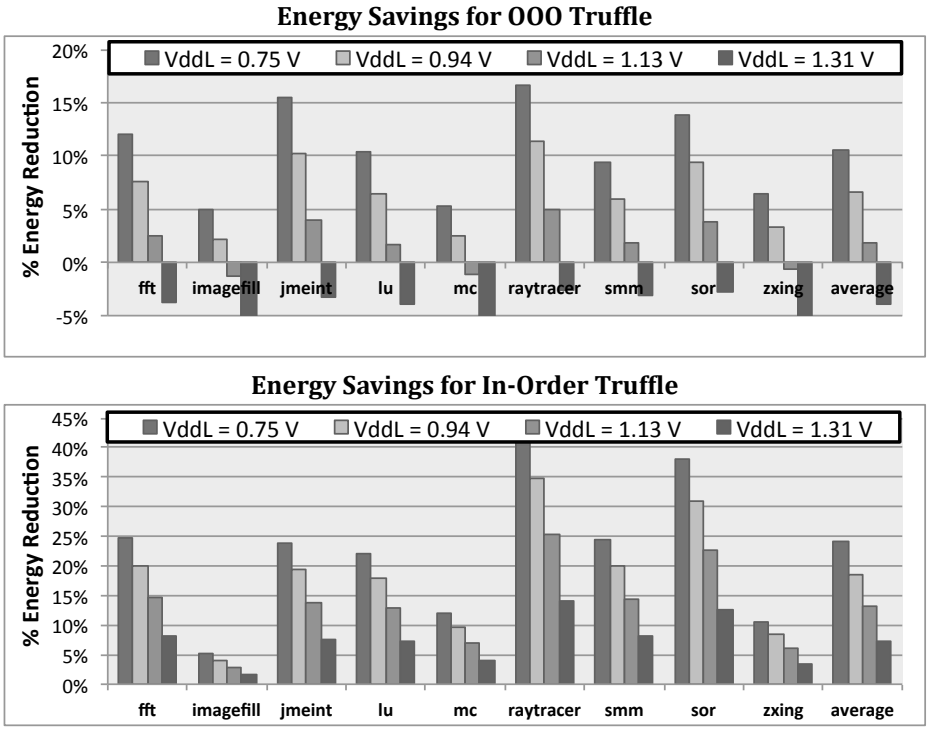
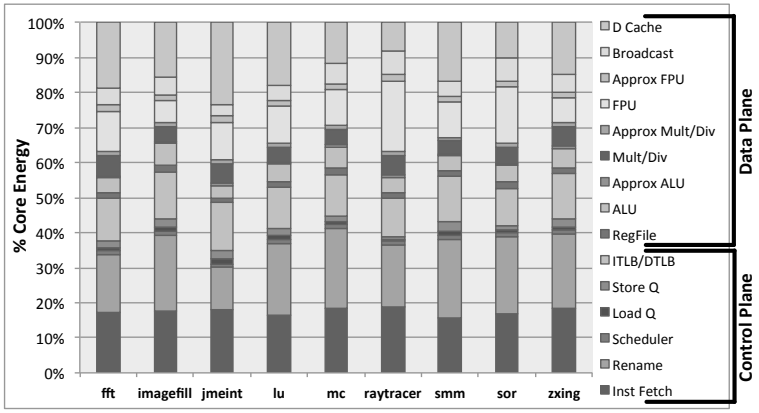


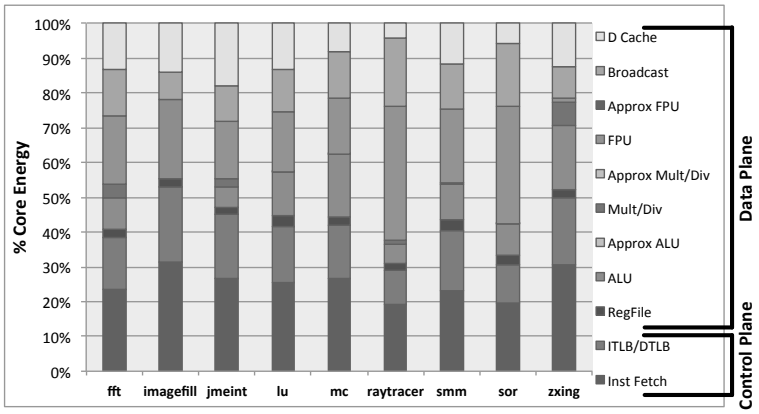
Figure 4.5: Percent energy reduction with unchecked OOO and in-order Truffle designs for various $V_{dd}L$ voltages.

instruction fetch and decode, register renaming, instruction issue and scheduling, load and store queues, and DTLB/ITLB, whereas in the in-order setting it includes only instruction fetch and decode and the TLBs. Since approximation helps reduce energy consumption in the data movement/processing plane only, the impact of Truffle in in-order cores is much higher. Furthermore, the OOO Truffle core is an aggressive four-wide multiple-issue processor whereas the in-order Truffle core is two-wide. Anything that can reduce the energy consumption of the instruction control plane indirectly helps increase Truffle's impact.

Figure 4.6 depicts the energy breakdown between different microarchitectural components in the OOO and in-order Truffle cores when $V_{dd}L = V_{dd}H$ (i.e., with fully-precise computation). Among the benchmarks, imagefill shows similar benefits for both designs. For this benchmark, 42% and 47% of the energy is consumed in the data movement/processing plane of the OOO and in-order Truffle cores, respectively. On the other hand, raytracer shows the largest difference in energy reduction between the two designs; here, the data movement/processing plane consumes



(a)



(b)

Figure 4.6: Percent energy consumed by different microarchitectural components in the (a) OOO and (b) in-order Truffle.

71% of the energy in the OOO core but just 50% in the in-order core. In summary, the simpler the instruction control plane in the processor, the higher the potential for energy savings with Truffle.

In addition to the design style of the Truffle core, the energy savings are dependent on the proportion of approximate computation in the execution. Figure 4.7 shows the percentage of approximate dynamic instructions along with the percentage of approximate ALU, multiply/divide, and floating point operations as well as the percentage of approximate data cache accesses. Among the benchmarks, imagefill has the lowest percentage of approximate instructions (20%) and no approximate floating point or multiplication operations—only 11% of its integer operations are approximate. As a fully integer application, it exhibits no opportunity for floating-point approximation, and its approximate integer operations are dwarfed by precise control-flow operations. imagefill also

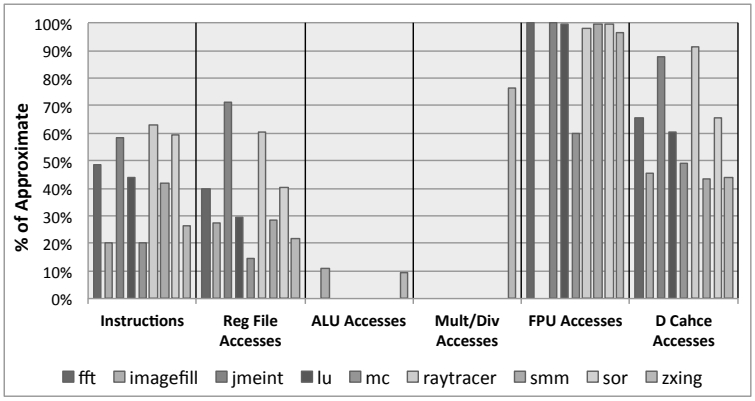


Figure 4.7: Percentage of approximate events.

has a low ratio of approximate data cache accesses, 45%. These characteristics result in low potential for Truffle, about 5% energy savings for $V_{dd}L = 0.75$ V. Conversely, raytracer shows the highest ratio of approximate instructions in the group. Nearly all (98%) of its floating point operations are approximate. In addition, raytracer has the highest ratio of approximate data cache accesses in the benchmark set, 91%, which makes it benefit the most from Truffle. The high rate of floating-point approximation is characteristic of the FP-dominated benchmarks we examined: for many applications, more than 90% of the FP operations are approximate. This is commensurate with the inherently approximate nature of FP representations. Furthermore, for many benchmarks, FP data constitutes the application's error-resilient data plane while integers dominate its error-sensitive control plane.

These results show that, as the proportion of approximate computation increases, the energy reductions from the Truffle microarchitecture also increase. Furthermore, some applications leave certain microarchitectural components unexercised, suggesting that higher error rates may be tolerable in those components. For example, none of the benchmarks except imagefill exercise the approximate integer ALU, and the approximate multiply/divide unit is not exercised at all. As a result, higher error rates in those components may be tolerable. The results also support the utility of application-specific $V_{dd}L$ settings, since each of the benchmarks exercise each approximate component differently.

Overall, these results show that disciplined approximation has great potential to enable low-power microarchitectures. Also, as expected, the simpler the microarchitecture, the higher the energy savings potential.

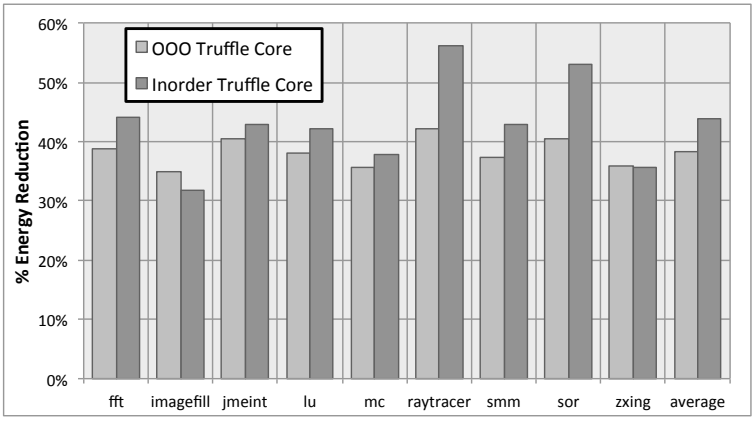


Figure 4.8: Percent energy reduction *potential* for checked in-order and OOO Truffle designs with $V_{dd}L = 0.75$ V.

Overheads. We modified McPAT and CACTI to model the overheads of Truffle as described in Section 4.4.4. The energy across all the benchmarks increases by at most 2% when the applications are compiled with no approximate instructions. The approximate functional units are power-gated when there are no approximate instructions in flight. The energy increase is due to the extra precision state per cache line and per register along with other microarchitectural changes. CACTI models show an increase of 3% in register file area due to the precision column and a 1% increase in the area of the level-1 data cache. The extra approximate functional units also contribute to the area overhead of Truffle.

4.5.3 Opportunities in a Checked Design

As discussed above, reducing energy consumption of the instruction control plane (and the energy used in *precise* instructions) can increase the overall impact of Truffle. Section 4.3.2 outlines a design that uses a sub-critical voltage and error detection/correction for the microarchitectural structures that need to behave precisely. We now present a simple limit study of the potential of such a design. Figure 4.8 presents the energy savings potential when the voltage level of the instruction control plane is reduced to 1.2 V, beyond the reliable voltage level of $V_{min} = 1.5$ V, and $V_{dd}L = 0.75$ V. The results show only the ideal case in which there is no penalty associated with error checking and correction in the precise computation. As illustrated, the gap in energy savings potential between the OOO and in-order designs is significantly reduced. In one case, imagefill, the checked OOO Truffle core shows higher potential compared to the checked in-order Truffle core. In

this benchmark, the energy consumption of the instruction control plane is more dominant in the OOO design and thus lower voltage for that plane is more effective than in the in-order design. Note that in an actual design, energy savings will be restricted by the error rates in the instruction control plane and the rate at which the precise instructions fail, triggering error recovery. The overhead of the error-checking structures will further limit the savings.

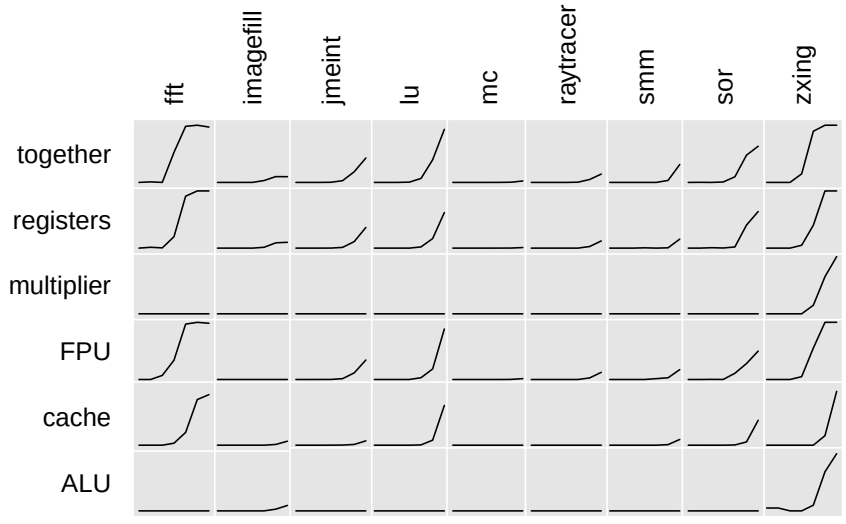
4.5.4 Error Propagation from Circuits to Applications

We now present a study of application QoS degradation as we inject errors in each of the microarchitectural structures that support approximate behavior. The actual pattern of errors caused by voltage reduction is highly design-dependent. Modeling the error distributions of approximate hardware is likely to involve guesswork; the most convincing evaluation of error rates would come from experiments with real Truffle hardware. For the present evaluation, we thoroughly explore a space of error rates in order to characterize the range of possibilities for the impact of approximation.

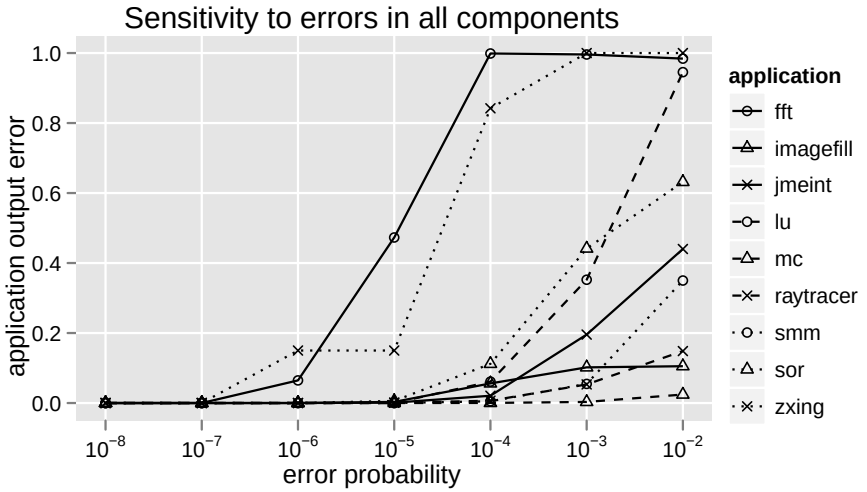
Figure 4.9 shows each benchmark's sensitivity to circuit-level errors in each microarchitectural component. Some applications are significantly sensitive to error injection in most components (fft, for example); others show very little degradation (imagefill, raytracer, mc, smm). Errors in some components tend to cause more application-level errors than others—for example, errors in the integer functional units (ALU and multiplier) only cause output degradation in the benchmarks with significant approximate integer computation (imagefill and zxing).

The variability in application sensitivity highlights again the utility of using a tunable $V_{dd}L$ to customize the architecture's error rate on a per-application basis (see Section 4.3.3). Most applications exhibit a critical error rate at which the application's output quality drops precipitously—for example, in Figure 4.9(b), fft exhibits low output error when all components have error probability 10^{-6} but significant degradation occurs at probability 10^{-5}. A software-controllable $V_{dd}L$ could allow each application to run at its lowest allowable power while maintaining acceptable output quality.

In general, the benchmarks do not exhibit drastically different sensitivities to errors in different components. A given benchmark that is sensitive to errors in the register file, for example, is also likely to be sensitive to errors in the cache and functional units.



(a)



(b)

Figure 4.9: Application sensitivity to circuit-level errors. Each cell in (a) has the same axes as (b): application QoS degradation is related to architectural error probability (on a log scale). The grid (a) shows applications' sensitivity to errors in each component in isolation; the row labeled "together" corresponds to experiments in which the error probability for all components is the same. The plot (b) shows these "together" configurations in more detail. The output error is averaged over 20 replications.

4.6 Concluding Remarks

The work in this chapter showed how von Neumann architectures can effectively trade accuracy for energy at the granularity of single instructions. We propose an ISA that simplifies the hardware by relying on the compiler to provide certain invariants statically, eliminating the need for checking or recovery at run time. We describe a high-level microarchitecture that supports interleaved high- and low-voltage operations and a detailed design for a dual-voltage SRAM array that implements approximation-aware caches and registers. Out dual-voltage microarchitectures can realize these energy savings by providing both approximate and precise computation and storage to be controlled at a fine grain by the compiler. We model the power of our proposed dual-voltage microarchitecture and evaluate its energy consumption in the context of a variety of error-tolerant benchmark applications. Experimental results show energy savings up to 43%; under reasonable assumptions, these benchmarks exhibit low or negligible degradation in output quality. However, the benefits are limited due to fact that the instruction steering plane of the processor needs to be always precise. In addition, the overhead of instruction-by-instruction execution can not be mitigated in this model of computation even when a subset of instructions are approximated.

Chapter 5

From a von Neumann Model to a Hybrid von Neumann-Neural Model of Computing

To enable much larger performance and efficiency gains achievable with approximate von Neumann processors, this chapter proposes an approach that leverages a simple programmer annotation ("approximable") to transform a hot code region from a von Neumann model to a neural model. We propose the Parrot algorithmic transformation that selects and trains a neural network to mimic a region of code. After the learning phase, the compiler replaces the original code with an invocation of a low-power accelerator called a neural processing unit (NPU). The NPU is tightly coupled to the processor's speculative pipeline to accelerate even small code regions. Since neural networks produce inherently approximate results, we define a simple programming model that allows programmers to identify approximable code regions—code that can produce imprecise but acceptable results. Offloading approximable code regions to NPUs is not only faster; it also is more energy efficient. The quality of the results with the Parrot transformation is commensurate with other work on quality trade-offs. The key idea in our approach is to learn how an approximable region of code behaves and replace the original code with an efficient computation of the learned model. One of the most important findings of our work is that neural networks can learn the behavior of regions of imperative code. This, in turn, enables a new class of efficient and low-complexity trainable accelerators based on neural networks. This chapter solves three concrete problems to enable this new class of accelerators: (a) fitting the Parrot algorithmic transformation into a conventional programing model; (b) exploring the space of neural network topologies and automatically selecting the appropriate neural configuration for each code region; (c) integrating a neural accelerator with a conventional processor pipeline. Our work, as the first instance of trainable accelerators, demonstrates opportunity for research both on learning techniques that can replace regions of imperative code and on hardware accelerators for these machine-learning models. We proposed a new class of accelerators that shows significant gains both in performance and energy when the abstraction of full accuracy is relaxed in general-purpose computing.
This chapter is based on work presented in MICRO (2012) [40] and IEEE Micro Top Picks (2013) [42].
This work is a result of collaboration with Adrian Sampson[a], Luis Ceze[a], and Doug Burger[b].

[a] University of Washington

[b] Microsoft Research

5.1 Introduction

The cessation of Dennard scaling has limited recent improvements in transistor speed and energy efficiency, resulting in slowed general-purpose processor improvements. Consequently, architectural innovation has become crucial to achieve performance and efficiency gains [32]. Furthermore, energy efficiency is a primary concern in computer systems.

However, there is a well-known tension between efficiency and programmability. Recent work has quantified three orders of magnitude of difference in efficiency between general-purpose processors and ASICs [56]. Since designing ASICs for the massive base of quickly changing, general-purpose applications is currently infeasible, practitioners are increasingly turning to programmable accelerators such as GPUs and FPGAs. Programmable accelerators provide an intermediate point between the efficiency of ASICs and the generality of conventional processors, gaining significant efficiency for restricted domains of applications.

Programmable accelerators exploit some characteristic of an application domain to achieve efficiency gains at the cost of generality. For instance, FPGAs exploit copious, fine-grained, and irregular parallelism but perform poorly when complex and frequent accesses to memory are required. GPUs exploit many threads and SIMD-style parallelism but lose efficiency when threads diverge. Emerging accelerators, such as BERET [52], Conservation Cores and QsCores [112, 113], or DySER [50], map regions of general-purpose code to specialized hardware units by leveraging either small, frequently-reused code idioms (BERET and DySER) or larger code regions amenable to hardware synthesis (Conservation Cores). Whether an application can use an accelerator effectively depends on the degree to which it exhibits the accelerator's required characteristics.

Tolerance to approximation is one such program characteristic that is growing increasingly important. Many modern applications—such as image rendering, signal processing, augmented reality, data mining, robotics, and speech recognition—can tolerate inexact computation in substantial portions of their execution [98, 45, 24, 78]. This tolerance can be leveraged for substantial performance and energy gains.

This chapter introduces a new class of programmable accelerators that exploit approximation for better performance and energy efficiency. The key idea is to *learn* how an original region of approximable code behaves and replace the original code with an efficient computation of the learned

model. This approach contrasts with previous work on approximate computation that extends conventional microarchitectures to support selective approximate execution, incurring instruction bookkeeping overheads [1, 25, 40, 80], or requires vastly different programming paradigms [15, 60, 76, 87]. Like emerging flexible accelerators [52, 112, 113, 50], our technique automatically offloads code segments from programs written in mainstream languages; but unlike prior work, it leverages changes in the semantics of the offloaded code.

We have identified three challenges that must be solved to realize effective trainable accelerators:

1. A **learning algorithm** is required that can accurately and efficiently mimic imperative code. We find that neural networks can approximate various regions of imperative code and propose the Parrot transformation, which exploits this finding (Section 5.2).

2. A **language and compilation framework** should be developed to transform regions of imperative code to neural network evaluations. To this end, we define a programming model and implement a compilation workflow to realize the Parrot transformation (Sections 5.3 and 5.4). The Parrot transformation starts from regions of approximable imperative code identified by the programmer, collects training data, explores the topology space of neural networks, trains them to mimic the regions, and finally replaces the original regions of code with trained neural networks.

3. An **architectural interface** is necessary to call a neural processing unit (NPU) in place of the original code regions. The NPU we designed is tightly integrated with a speculative out-of-order core. The low-overhead interface enables acceleration even when fine-grained regions of code are transformed. The core communicates both the neural configurations and runtime invocations to the NPU through extensions to the ISA (Sections 5.5 and 5.6).

Rather than contributing a new design for neural network implementation, this chapter presents a new technique for harnessing hardware neural networks in general-purpose computations. We show that using neural networks to replace regions of imperative code is feasible and profitable by experimenting with a variety of applications, including FFT, gaming, clustering, and vision algorithms (Section 5.7). These applications do not belong to the class of modeling and prediction

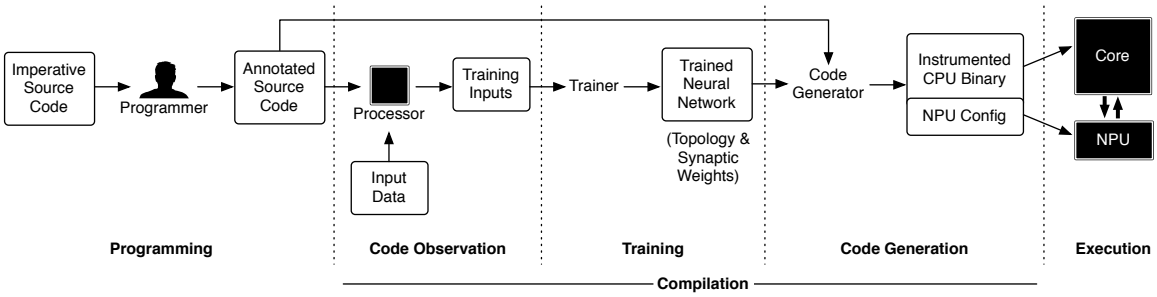


Figure 5.1: The Parrot transformation at a glance: from annotated code to accelerated execution on an NPU-augmented core.

that typically use neural networks. For each application, we identify a single approximable function that dominates the program's execution time. NPU acceleration provides 2.3× average whole-application speedup and 3.0× average energy savings for these benchmarks with average accuracy greater than 90% in all cases. NPUs provide a new class of accelerators—with implementation potential in both analog and digital domains—for emerging approximate applications.

5.2 Overview

The *Parrot transformation* is an algorithmic transformation that converts regions of imperative code to neural networks. Because neural networks expose considerable parallelism and can be efficiently accelerated using dedicated hardware, the Parrot transformation can yield significant performance and energy improvements. The transformation uses a training-based approach to produce a neural network that approximates the behavior of candidate code. A transformed program runs primarily on the main core and invokes an auxiliary hardware structure, the neural processing unit (NPU), to perform neural evaluation instead of executing the replaced code. Figure 5.1 shows an overview of our proposed approach, which has three key phases: programming, in which the programmer marks code regions to be transformed; compilation, in which the compiler selects and trains a suitable neural network and replaces the original code with a neural network invocation; and execution.

Programming. During development, the programmer explicitly annotates functions that are amenable to approximate execution and therefore candidates for the Parrot transformation. Because tolerance of approximation is a semantic property, it is the programmer's responsibility to select code whose approximate execution would not compromise the overall reliability of the application. This

is common practice in the approximate computing literature [98, 40, 25]. We discuss our programming model in detail in Section 5.3.

Compilation. Once the source code is annotated, as shown in Figure 5.1, the compiler applies the Parrot transformation in three steps: (1) code observation; (2) neural network selection and training; and (3) binary generation. Section 5.4 details these steps.

In the code observation step, the compiler observes the behavior of the candidate code region by logging its inputs and outputs. This step is similar to profiling. The compiler instruments the program with probes on the inputs and outputs of the candidate functions. Then, the instrumented program is run using representative input sets such as those from a test suite. The probes log the inputs and outputs of the candidate functions. The logged input–output pairs constitute the training and validation data for the next step.

The compiler uses the collected input–output data to configure and train a neural network that mimics the candidate region. The compiler must discover the topology of the neural network as well as its synaptic weights. It uses the backpropagation algorithm [97] coupled with a topology search (see Section 5.4.2) to configure and train the neural network.

The final step of the Parrot transformation is code generation. The compiler first generates a configuration for the NPU that implements the trained neural network. Then, the compiler replaces each call to the original function with a series of special instructions that invoke the NPU, sending the inputs and receiving the computed outputs. The NPU configuration and invocation is performed through ISA extensions that are added to the core.

Execution. During deployment, the transformed program begins execution on the main core and configures the NPU. Throughout execution, the NPU is invoked to perform a neural network evaluation in lieu of executing the original code region. The NPU is integrated as a tightly-coupled accelerator in the processor pipeline. Invoking the NPU is faster and more energy-efficient than executing the original code region, so the program as a whole is accelerated.

Many NPU implementations are feasible, from all-software to specialized analog circuits. Because the Parrot transformation's effectiveness rests on the efficiency of neural network evaluation, it is essential that invoking the NPU be fast and low-power. Therefore, we describe a high-

performance hardware NPU design based on a digital neural network ASIC (Section 5.6) and architecture support to facilitate low-latency NPU invocations (Section 5.5).

A key insight in this chapter is that it is possible to automatically discover and train neural networks that effectively approximate imperative code from diverse application domains. These diverse applications do not belong to the class of modeling and prediction applications that typically use neural networks. *As Figure 5.4a illustrates, the Parrot transformation convert divers regions of code to a common representation, i.e. neural networks. Using neural networks as a common representation enables a novel use of hardware neural networks to accelerate many approximate applications.*

5.3 Programming Model

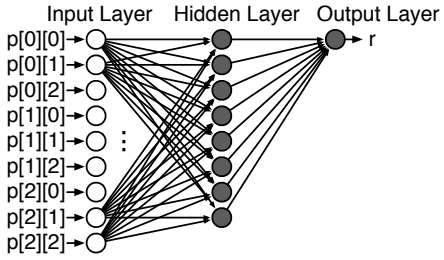
The Parrot transformation starts with the programmer identifying candidate code regions. These candidate regions need to comply with certain criteria to be suitable for the transformation. This section discusses these criteria as well as the concrete language interface exposed to the programmer. After the candidate regions are identified, the Parrot transformation is fully automated.

5.3.1 Code Region Criteria

Candidate code for the Parrot transformation must satisfy three criteria: it must be frequently executed (i.e., a "hot" function); it must tolerate imprecision in its computation; and it must have well-defined inputs and outputs.

Hot code. Like any acceleration technique, the Parrot transformation should replace hot code. The Parrot transformation can be applied to a wide range of code from small functions to entire algorithms. The code region can contain function calls, loops, and complex control flow whose cost can be elided by the Parrot transformation. When applied to smaller regions of code, the overhead of NPU invocation needs to be low to make the transformation profitable. A traditional performance profiler can reveal hot code.

For example, edge detection is a widely applicable image processing computation. Many implementations of edge detection use the Sobel filter, a 3×3 matrix convolution that approximates the image's intensity gradient. As the bottom box in Figure 5.2a shows, the local Sobel filter com-



(b) sobel transformed to a $9 \rightarrow 8 \rightarrow 1$ NN

```
1  float sobel [[PARROT]] (float[3][3] p){
     float x, y, r;
3    x = (p[0][0] + 2 * p[0][1] + p[0][2]);
     x = (p[2][0] + 2 * p[2][1] + p[2][2]);
5    y = (p[0][2] + 2 * p[1][2] + p[2][2]);
     y = (p[0][0] + 2 * p[1][1] + p[2][0]);
7    r = sqrt(x * x + y * y);
     if (r >= 0.7071) r = 0.7070;
9    return r;
   }
```

```
   void edgeDetection(
2    Image& srcImg,
     Image& dstImg
4  ){
     float[3][3] p;
6    float pixel;

8    for(int y = 0; y < srcImg.height; ++y)
       for(int x = 0; x < srcImg.width; ++x)
10       srcImg.toGrayeScale(x,y);

12   for(int y = 0; y < srcImg.height; ++y)
       for(int x = 0; x < scrImg.width; ++x){
14       p = srcImg.build3x3Window(x, y);
         pixel = sobel(p);
16       dstImg.setPixel(x, y, pixel);
       }
18 }
```

(a) Original implementation of the Sobel filter

```
   void edgeDetection(
2    Image& srcImg,
     Image& dstImg
4  ){
     float[3][3] p;
6    float pixel;

8    for(int y = 0; y < srcImg.height; ++y)
       for(int x = 0; x < srcImg.width; ++x)
10       srcImg.toGrayeScale(x,y);

12   for(int y = 0; y < srcImg.height; ++y)
       for(int x = 0; x < scrImg.width; ++x){
14       p = srcImg.build3x3Window(x, y);
         NPU_SEND(p[0][0]); NPU_SEND(p[0][1]);
16       NPU_SEND(p[0][2]); NPU_SEND(p[1][0]);
         NPU_SEND(p[1][1]); NPU_SEND(p[1][2]);
18       NPU_SEND(p[2][0]); NPU_SEND(p[2][1]);
         NPU_SEND(p[2][2]);
20
         NPU_RECEIVE(pixel);
22
         dstImg.setPixel(x, y, pixel);
24     }
   }
```

(c) Parrot-transformed code; an NPU invocation replaces the function call

Figure 5.2: Three stages in the transformation of an edge detection using the Sobel filter.

putation (the sobel function) is executed many times during edge detection, so the convolution is a hot function in the overall algorithm and a good candidate for the Parrot transformation.

Approximability. Code regions identified for the Parrot transformation will behave approximately during execution. Therefore, programs must incorporate application-level tolerance of imprecision. This requires the programmer to ensure that imprecise results from candidate regions will not cause catastrophic failures. As prior work on approximate programming [98, 25, 4, 80, 101] has shown, it is not difficult to deem regions approximable.

Beyond determining that a code region may safely produce imprecise results, the programmer need not reason about the mapping between the code and a neural network. While neural networks are more precise for some functions than they are for others, we find that they can accurately mimic many functions from real programs (see Section 5.7). Intuitively, however, they are less likely to effectively approximate chaotic functions, in which even large training sets can fail to capture enough of the function's behavior to generalize to new inputs. However, the efficacy of neural network approximation can be assessed empirically. The programmer should annotate all approximate code; the compiler can then assess the accuracy of a trained neural network in replacing each function and select only those functions for which neural networks are a good match.

In the Sobel filter example, parts of the code that process the pixels can be approximated. The code region that computes pixel addresses and builds the window for the sobel function (line 8 in the bottom box of Figure 5.2a) needs to be precise to avoid memory access violations. However, the sobel function, which estimates the intensity gradient of a pixel, is fundamentally approximate. Thus, approximate execution of this function will not result in catastrophic failure and, moreover, is unlikely to cause major degradation of the overall edge detection quality. These properties make the sobel function a suitable candidate region for approximate execution.

Well-defined inputs and outputs. The Parrot transformation replaces a region of code with a neural network that has a fixed number of inputs and outputs. Therefore, it imposes two restrictions on the code regions that can feasibly be replaced. First, the inputs to and outputs from the candidate region must be of a fixed size known at compile time. For example, the code may not dynamically write an unbounded amount of data to a variable-length array. Second, the code must be *pure:* it

must not read any data other than its inputs nor affect any state other than its outputs (e.g., via a system call). These two criteria can be checked statically.

The sobel function in Figure 5.2a complies with these requirements. It takes nine statically identifiable floating-point numbers as input, produces a single output, and has no side effects.

5.3.2 Annotation

In this work, we apply the Parrot transformation to entire functions. To identify candidate functions, the programmer marks them with an annotation (e.g., using C++11 [[annotation]] syntax as shown in Figure 5.2a). The programmer is responsible for ensuring that the function has no side effects, reads only its arguments, and writes only its return value. Each argument type and the return type must have a fixed size. If any of these types is a pointer type, it must point to a fixed-size value; this referenced value is then considered the neural network input or output rather than the pointer itself. If the function needs to return multiple values, it can return a fixed-size array or a C struct. After the programmer annotates the candidate functions, the Parrot transformation is completely automatic and transparent: no further programmer intervention is necessary.

Other annotation approaches. Our current system depends on explicit programmer annotations at the granularity of functions. While we find that explicit function annotations are straightforward to apply (see Section 5.7), static analysis techniques could be used to further simplify the annotation process. For example, in an approximation-aware programming language such as EnerJ [98], the programmer uses type qualifiers to specify which data is non-critical and may be approximated. In such a system, the Parrot transformation can be automatically applied to any block of code that only affects approximate data. That is, the candidate regions for the Parrot transformation would be implicitly defined.

Like prior work on approximate computing, we acknowledge that some programmer guidance is essential when identifying error-tolerant code [98, 80, 25, 40, 4]. Tolerance to approximation is an inherently application-specific property. Fortunately, language-level techniques like EnerJ demonstrate that the necessary code annotations can be intuitive and straightforward for programmers to apply.

5.4 Compilation Workflow

Once the program has been annotated, the compilation workflow implements the Parrot transformation in three steps: observation, training, and instrumented binary generation.

5.4.1 Code Observation

In the first phase, the compiler collects input–output pairs for the target code that reflect real program executions. This in-context observation allows the compiler to train the neural network on a realistic data set. The compiler produces an instrumented binary for the source program that includes probes on the input and output of the annotated function. Each time the candidate function executes, the probes record its inputs and outputs. The program is run repeatedly using test inputs. The output of this phase is a training data set: each input–output pair represents a sample for the training algorithm. The system also measures the minimum and maximum value for each input and output; the NPU normalizes values using these ranges during execution.

The observation phase resembles the profiling runs used in profile-guided compilation. Specifically, it requires representative test inputs for the application. The inputs may be part of an existing test suite or randomly generated. In many cases, a small number of application test inputs are sufficient to train a neural network because the candidate function is executed many times in a single application run. In our edge detection example, the sobel function runs for every pixel in the input image. So, as Section 5.7 details, training sobel on a single 512×512 test image provides 262144 training data points and results in acceptable accuracy when computing on unseen images.

Although we do not explore it in the dissertation, automatic input generation could help cover the space of possible inputs and thereby achieve a more accurate trained neural network. In particular, the compiler could synthesize new inputs by interpolating values between existing test cases.

5.4.2 Training

The compiler uses the training data to produce a neural network that replaces the original function. There are a variety of types of artificial neural networks in the literature, but we narrow the search space to multilayer perceptrons (MLPs) due to their broad applicability.

The compiler uses the backpropagation algorithm [97] to train the neural network. Backpropagation is a gradient descent algorithm that iteratively adjusts the weights of the neural network according to each input–output pair. The learning rate, a value between 0 and 1, is the step size of the gradient descent and identifies how much a single example affects the weights. Since backpropagation on MLPs is not convex and the compilation procedure is automatic, we choose a small learning rate of 0.01. Larger steps can cause oscillation in the training and prevent convergence. One complete pass over the training data is called an epoch. Since the learning rate is small, the epoch count should be large enough to ensure convergence. Empirically, we find that 5000 epochs achieve a good balance of generalization and accuracy. Larger epoch counts can cause overtraining and adversely affect the generalization ability of the network while smaller epoch counts may result in poor accuracy.

Neural network topology selection. In addition to running backpropagation, this phase selects a network topology that balances between accuracy and efficiency. An MLP consists of a fully-connected set of neurons organized into layers: the input layer, any number of "hidden" layers, and the output layer (see Figure 5.2b). A larger, more complex network offers better accuracy potential but is likely to be slower and less power-efficient than a small, simple neural network.

To choose the topology, we use a simple search algorithm guided by the mean squared error of the neural network when tested on an unseen subset of the observed data. The error evaluation uses a typical cross-validation approach: the compiler partitions the data collected during observation into a *training set*, 70% of the observed data, and a *test set*, the remaining 30%. The topology search algorithm trains many different neural network topologies using the training set and chooses the one with the highest accuracy on the test set and the lowest latency on the NPU (prioritizing accuracy).

The space of possible topologies is large, so we restrict the search to neural networks with at most two hidden layers. We also limit the number of neurons per hidden layer to powers of two up to 32. (The numbers of neurons in the input and output layers are predetermined based on the number of inputs and outputs in the candidate function.) These choices limit the search space to 30 possible topologies. The maximum number of hidden layers and maximum neurons per hidden

layer are compilation options and can be specified by the user. Although the candidate topologies can be trained in parallel, enlarging the search space increases the compilation time.

The output from this phase consists of a neural network topology—specifying the number of layers and the number of neurons in each layer—along with the weight for each neuron and the normalization range for each input and output. Figure 5.2b shows the three-layer MLP that replaces the sobel function. Each neuron in the network performs a weighted sum on its inputs and then applies a sigmoid function to the result of weighted sum.

On-line training. Our present system performs observation and training prior to deployment; an alternative design could train the neural network concurrently with in-vivo operation. On-line training could improve accuracy but would result in runtime overheads. To address these overheads, an on-line training system could offload neural network training and configuration to a remote server. With off-site training, multiple deployed application instances could centralize their training to increase input space coverage.

5.4.3 Code Generation

After the training phase, the compiler generates an instrumented binary that runs on the core and invokes the NPU instead of calling the original function. The program configures the NPU when it is first loaded by sending the topology parameters and synaptic weights to the NPU via its configuration interface (Section 5.6.2). The compiler replaces the calls to the original function with special instructions that send the inputs to the NPU and collect the outputs from it. The configuration and input–output communication occurs through ISA extensions discussed in Section 5.5.1.

5.5 Architecture Design for NPU Acceleration

Since candidate regions for the Parrot transformation can be fine-grained, NPU invocation must be low-overhead to be beneficial. Ideally, the NPU should integrate tightly with the processor pipeline. The processor ISA also needs to be extended to allow programs to configure and invoke the NPU during execution. Moreover, NPU invocation should not prevent speculative execution. This section

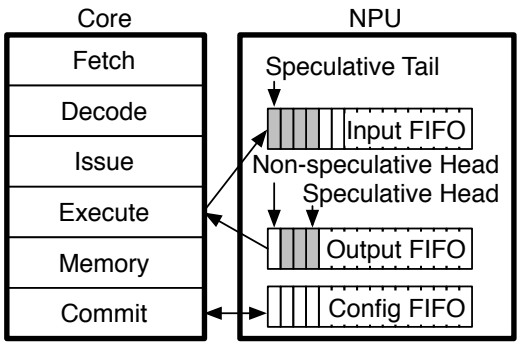


Figure 5.3: The NPU exposes three FIFO queues to the core. The speculative state of the FIFOs is shaded.

discusses the ISA extensions and microarchitectural mechanism for tightly integrating the NPU with an out-of-order processor pipeline.

5.5.1 ISA Support for NPU Acceleration

The NPU is a variable-delay, tightly-coupled accelerator that communicates with the rest of the core via FIFO queues. As shown in Figure 5.3, the CPU–NPU interface consists of three queues: one for sending and retrieving the configuration, one for sending the inputs, and one for retrieving the neural network's outputs. The ISA is extended with four instructions to access the queues. These instructions assume that the processor is equipped with a single NPU; if the architecture supports multiple NPUs or multiple stored configurations per NPU, the instructions may be parameterized with an operand that identifies the target NPU.

- **enq.c %r**: enqueues the value of the register r into the config FIFO.
- **deq.c %r**: dequeues a configuration value from the config FIFO to the register r.
- **enq.d %r**: enqueues the value of the register r into the input FIFO.
- **deq.d %r**: dequeues the head of the output FIFO to the register r.

To set up the NPU, the program executes a series of enq.c instructions to send configuration parameters—number of inputs and outputs, network topology, and synaptic weights—to the NPU. The operating system uses deq.c instructions to save the NPU configuration during context switches. To invoke

the NPU, the program executes enq.d repeatedly to send inputs to the configured neural network. As soon as all of the inputs of the neural network are enqueued, the NPU starts computation and puts the results in its output FIFO. The program executes deq.d repeatedly to retrieve the output values.

Instead of special instructions, an alternative design could use memory-mapped IO to communicate with the NPU. This design would require special fence instructions to prevent interference between two consecutive invocations and could impose a large overhead per NPU invocation.

5.5.2 Speculative NPU-Augmented Architecture

Scheduling and issue. To ensure correct communication with the NPU, the processor must issue NPU instructions in order. To accomplish this, the renaming logic implicitly considers every NPU instruction to read and write a designated "dummy" architectural register. The scheduler will therefore treat all NPU instructions as dependent. Furthermore, the scheduler only issues an enqueue instruction if the corresponding FIFO is not full. Similarly, a dequeue instruction is only issued if the corresponding FIFO is not empty.

Speculative execution. The processor can execute enq.d and deq.d instructions speculatively. Therefore, the head pointer of the input FIFO can only be updated—and consequently the entries recycled—when: (1) the enqueue instruction commits; and (2) the NPU finishes processing that input. When an enq.d instruction reaches the commit stage, a signal is sent to the NPU to notify it that the input FIFO head pointer can be updated.

To ensure correct speculative execution, the output FIFO maintains two head pointers: a speculative head and a non-speculative head. When a dequeue instruction is issued, it reads a value from the output FIFO and the speculative head is updated to point to the next output. However, the non-speculative head is not updated to ensure that the read value is preserved in case the issue of the instruction was a result of misspeculation. The non-speculative head pointer is only updated when the instruction commits, freeing the slot in the output FIFO.

In case of a flush due to branch or dependence misspeculation, the processor sends the number of squashed enq.d and deq.d instructions to the NPU. The NPU adjusts its input FIFO tail pointer and output FIFO speculative head pointer accordingly. The NPU also resets its internal control state

if it was processing any of the invalidated inputs and adjusts the output FIFO tail pointer to invalidate any outputs that are based on the invalidated inputs. The rollback operations are performed concurrently for the input and output FIFOs.

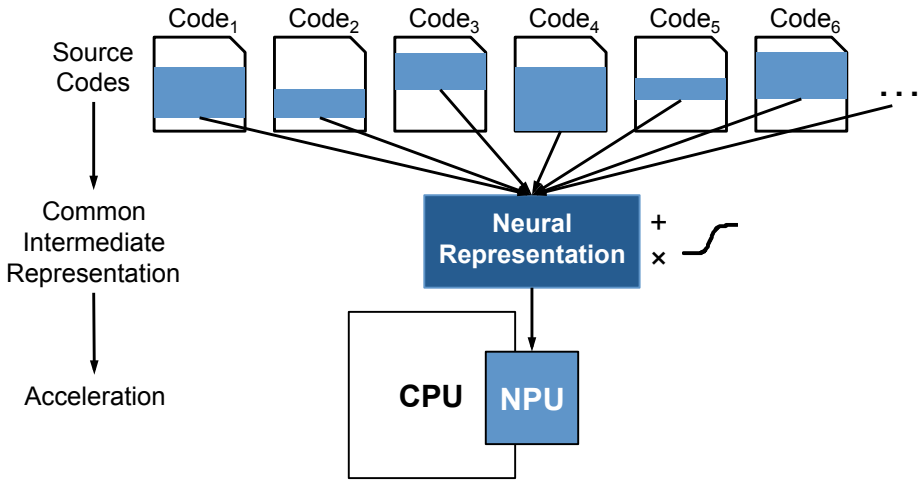
The enq.c and deq.c instructions, which are only used to read and write the NPU configuration, are not executed speculatively.

Interrupts. If an interrupt were to occur during an NPU invocation, the speculative state of the NPU would need to be flushed. The remaining non-speculative data in the input and output FIFOs would need to be saved and then restored when the process resumes. One way to avoid this complexity is to disable interrupts during NPU invocations; however, this approach requires that the invocation time is finite and ideally short as to not delay interrupts for too long.

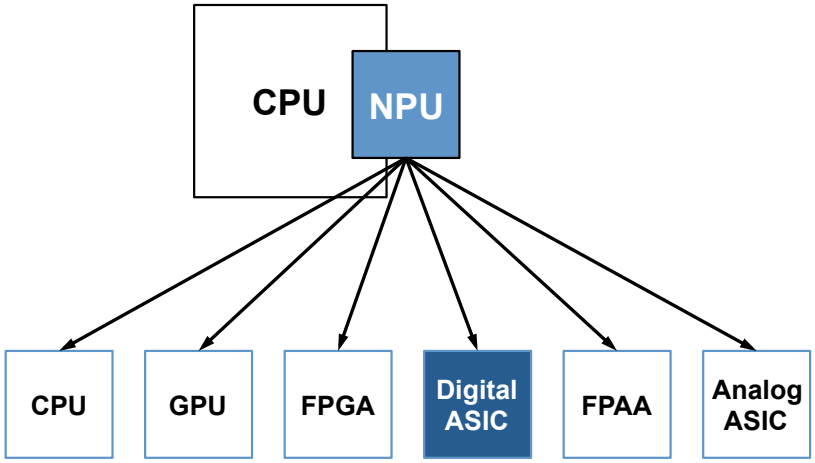
Context switches. The NPU's configuration is architectural state, so the operating system must save and restore the configuration data on a context switch. The OS reads out the current NPU configuration using the deq.c instruction and stores it for later reconfiguration when the process is switched back in. To reduce context switch overheads, the OS can use the same lazy context switch techniques that are typically used with floating point units [88].

5.6 Neural Processing Unit

There are many implementation options for NPUs with varying trade-offs in performance, power, area, and complexity, as illustrated by Figure 5.4. At one extreme are software implementations running on a CPU or GPU [90, 54]. Since these implementations have higher computation and communication overheads, they are likely more suitable for very large candidate regions, when the invocation cost can be better amortized. Next on the scale are FPGA-based implementations [119]. Digital ASIC designs are likely to be lower-latency and more power-efficient than FPGA-based implementations [94, 31]. Since neural networks are themselves approximable, their implementation can also be approximate. Therefore, we can improve efficiency further and use approximate digital circuits (e.g., sub-critical voltage supply). In the extreme, one can even use custom analog circuitry or FPAAs [13, 100, 106]. In fact, we believe that analog NPUs have significant potential and we plan to explore them in future work. We focus on an ASIC design operating at the same critical voltage as

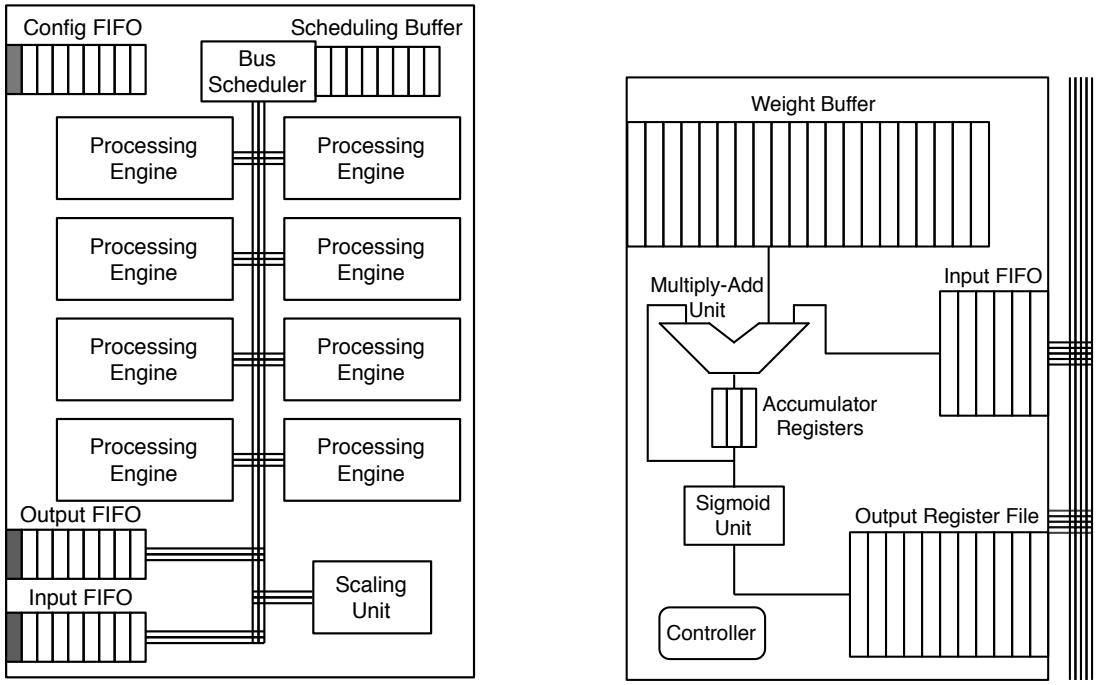


(a) Neural networks as a common intermediate representation



(b) Design alternatives for NPUs

Figure 5.4: (a) The Parrot algorithmic transformation converts different regions of code to a common neural intermediate representation. Neural networks as a common representation enable acceleration of diverse applications using a single NPU. (b) Design space of NPU implementations. This chapter focuses on a precise digital ASIC design.



(a) 8-PE NPU (b) Single processing engine

Figure 5.5: Reconfigurable 8-PE NPU.

the main core. This implementation represents a reasonable trade-off between efficiency and complexity; it is able to accelerate a wide variety of applications without the complexity of integrating analog or sub-critical components with the processor.

5.6.1 Reconfigurable Digital NPU

The Parrot transformation produces different neural network topologies for different code regions. Thus, we propose a reconfigurable NPU design that accelerates the evaluation of a range of neural topologies. As shown in Figure 5.5a, the NPU contains eight identical processing engines (PEs) and one scaling unit. Although the design can scale to larger numbers of PEs, we find that the speedup gain beyond 8 PEs is small (see Section 5.7). The scaling unit scales the neural network's inputs and outputs if necessary using scaling factors defined in the NPU configuration process.

The PEs in the NPU are statically scheduled. The scheduling information is part of the configuration information for the NPU, which is based on the neural network topology derived during the training process. In the NPU's schedule, each neuron in the neural network is assigned to one of

the eight PEs. The neural network's topology determines a static schedule for the timing of the PE computations, bus accesses, and queue accesses.

The NPU stores the bus scheduling information in its circular scheduling buffer (shown in Figure 5.5a). Each entry in this buffer schedules the bus to send a value from a PE or the input FIFO to a set of destination PEs or the output FIFO. Every scheduling buffer entry consists of a source and a destination. The source is either the input FIFO or the identifier of a PE along with an index into its output register file (shown in Figure 5.5b). The destination is either the output FIFO or a bit field indicating the destination PEs.

Figure 5.5b shows the internal structure of a single PE. Each PE performs the computation for all of its assigned neurons. Namely, because the NPU implements a sigmoid-activation multilayer perceptron, each neuron computes its output as $y = \text{sigmoid}(\sum_i (x_i \times w_i))$ where x_i is an input to the neuron and w_i is its corresponding weight. The weight buffer, a circular buffer, stores the weights. When a PE receives an input from the bus, it stores the value in its input FIFO. When the neuron weights for each PE are configured, they are placed into the weight buffer; the compiler-directed schedule ensures that the inputs arrive in the same order that their corresponding weights appear in the buffer. This way, the PE can perform multiply-and-add operations in the order the inputs enter the PE's input FIFO.

Each entry in the weight buffer is augmented by one bit indicating whether a neuron's multiply-add operation has finished. When it finishes, the PE applies the sigmoid function, which is implemented as a lookup table, and write the result to its output register file. The per-neuron information stored in the weight buffer also indicates which output register should be used.

5.6.2 NPU Configuration

During code generation (Section 5.4.3), the compiler produces an NPU configuration that implements the trained neural network for each candidate function. The static NPU scheduling algorithm first assigns an order to the inputs of the neural network. This order determines both the sequence of $\mathrm{enq.d}$ instructions that the CPU will send to the NPU during each invocation and the order of multiply-add operations among the NPU's PEs. Then, the scheduler takes the following steps for each layer of the neural network:

1. Assign each neuron to one of the processing engines.

2. Assign an order to the multiply-add operations considering the order assigned to the inputs of the layer.

3. Assign an order to the outputs of the layer.

4. Produce a bus schedule reflecting the order of operations.

The ordering assigned for the final layer of the neural network dictates the order in which the program will retrieve the NPU's output using deq.d instructions.

5.7 Evaluation

To evaluate the effectiveness of the Parrot transformation, we apply it to several benchmarks from diverse application domains. For each benchmark, we identify a region of code that is amenable to the Parrot transformation. We evaluate whole-application speedup and energy savings using cycle-accurate simulation and a power model. We also examine the resulting trade-off in computation accuracy. We perform a sensitivity analysis to examine the effect of NPU PE count and communication latency on the performance benefits.

5.7.1 Benchmarks and the Parrot Transformation

Table 5.1 lists the benchmarks used in this evaluation. These benchmarks are all written in C. The application domains—signal processing, robotics, gaming, compression, machine learning, and image processing—are selected for their usefulness to general applications and tolerance to imprecision. The domains are commensurate with evaluations of previous work on approximate computing [98, 40, 101, 1, 78, 80].

Table 5.1 also lists the input sets used for performance, energy, and accuracy assessment. These input sets are different from the ones used during the training phase of the Parrot transformation. For applications with random inputs we use a different random input set. For applications with image input, we use a different image.

Table 5.1: The benchmarks evaluated, characterization of each transformed function, training data, and the result of the Parrot transformation.

	Description	Type	Evaluation Input Set	# of Function Calls	# of Loops	# of ifs/ elses	# of x86-64 Instructions	Training Input Set	Neural Network Topology	NN MSE	Error Metric	Error
fft	Radix-2 Cooley-Tukey fast Fourier	Signal Processing	2048 Random Floating Point Numbers	2	0	0	34	32768 Random Floating Point Numbers	1 -> 4 -> 4 -> 2	0.00002	Average Relative Error	7.22%
inversek2j	Inverse kinematics for 2-joint arm	Robotics	10000 (x,y) Random Coordinates	4	0	0	100	10000 (x,y) Random Coordinates	2 -> 8 -> 2	0.00563	Average Relative Error	7.50%
jmeint	Triangle intersection detection	3D Gaming	10000 Random Pairs of 3D Triangle Coordinates	32	0	23	1,079	100000 Random Pairs of 3D Triangle Coordinates	18 -> 32 -> 8 -> 2	0.00530	Miss Rate	7.32%
jpeg	JPEG encoding	Compression	220x200-Pixel Color Image	3	4	0	1,257	Three 512x512-Pixel Color Images	64 -> 16 -> 64	0.00890	Image Diff	9.56%
kmeans	K-means clustering	Machine Learning	220x200-Pixel Color Image	1	0	0	26	50000 Pairs of Random (r, g, b) Values	6 -> 8 -> 4 -> 1	0.00169	Image Diff	6.18%
sobel	Sobel edge detector	Image Processing	220x200-Pixel Color Image	3	2	1	88	One 512x512-Pixel Color Image	9 -> 8 -> 1	0.00234	Image Diff	3.44%

Code annotation. The C source code for each benchmark was annotated as described in Section 5.3: we identified a single pure function with fixed-size inputs and outputs. No algorithmic changes were made to the benchmarks to accommodate the Parrot transformation. There are many choices for the selection of target code and, for some programs, multiple NPUs may even have been beneficial. For the purposes of this evaluation, however, we selected a single target region per benchmark that was easy to identify, frequently executed as to allow for efficiency gains, and amenable to learning by a neural network. Qualitatively, we found it straightforward to identify a reasonable candidate function in each benchmark.

Table 5.1 shows the number of function calls, conditionals, and loops in each transformed function. The table also shows the number of x86-64 instructions for the target function when compiled by GCC 4.4.6 at the -O3 optimization level. We do not include the statistics of the standard library functions in these numbers. In most of these benchmarks, the target code contains complex control flow including conditionals, loops, and method calls. In jmeint, the target code contains the bulk of the algorithm, including many nested method calls and numerous conditionals. In jpeg, the transformation subsumes the discrete cosine transform and quantization phases, which contain function calls and loops. In fft, inversek2j, and sobel, the target code consists mainly of arithmetic operations and simpler control flow. In kmeans, the target code is the 0 distance calculation, which is simple and fine-grained yet frequently executed. In each case, the target code is side-effect-free and the number of inputs/outputs are statically identifiable.

Training data. To train the NPU for each application, we have used either (1) typical program inputs (e.g., sample images) or (2) a limited number of random inputs. For the benchmarks that use random inputs, we determined the permissible range of parameters in the code and generated uniform random inputs in that range. For the image-based benchmarks, we used three standard images that are used to evaluate image processing algorithms (lena, mandrill, and peppers). For kmeans, we supplied random inputs to the code region to avoid overtraining on a particular test image. Table 5.1 shows the specific image or application input used in the training phase for each benchmark. We used different random inputs and different images for the final accuracy evaluation.

Neural networks. The "Neural Network Topology" column in Table 5.1 shows the topology of the trained neural network discovered by the training stage described in Section 5.4.2. The "NN MSE" column shows the mean squared error for each neural network on the test subset of the training data. For example, the topology for jmeint is $18 \rightarrow 32 \rightarrow 8 \rightarrow 2$, meaning that the neural network takes in 18 inputs, produces 2 outputs, and has two hidden layers with 32 and 8 neurons respectively. As the results show, the compilation workflow was able to find a neural network that accurately mimics each original function. However, different topologies are required to approximate different functions.

Different applications require different neural network topologies, so the NPU structure must be reconfigurable.

Output quality. We use an application-specific error metric, shown in Table 5.1, to assess the quality of each benchmark's output. In all cases, we compare the output of the original untransformed application to the output of the transformed application. For fft and inversek2j, which generate numeric outputs, we measure the average relative error. jmeint calculates whether two three-dimensional triangles intersect; we report the misclassification rate. For jpeg, kmeans, and sobel, which produce image outputs, we use the average root-mean-square image difference. The column labeled "Error" in Table 5.1 shows the whole-application error of each benchmark according to its error metric. Unlike the "NN MSE" error values, this application-level error assessment accounts for accumulated errors due to repeated execution of the transformed function.

Application average error rates range from 3% to 10%. This quality-of-service loss is commensurate with other work on quality trade-offs. Among hardware approximation techniques, Truffle [40] shows similar error (3–10%) for some applications and much greater error (above 80%) for others in a moderate configuration. The evaluation of EnerJ [98] also has similar error rates; two thirds of the applications exhibit error greater than 10% in the most energy-efficient configuration. Green [4], a software technique, has error rates below 1% for some applications but greater than 20% for others. A case study by Misailovic et al. [83] explores manual optimizations of a video encoder, x264, that trade off 0.5–10% quality loss.

Table 5.2: Microarchitectural parameters for the core, caches, memory, NPU, and each PE in the NPU.

Core	
Architecture	x86-64
Fetch/Issue Width	4/6
INT ALUs/FPUs	3/2
Load/Store FUs	2/2
ROB Entries	96
Issue Queue Entries	32
INT/FP Physical Registers	256/256
Branch Predictor	Tournament, 48 KB
BTB Sets/Ways	1024/4
RAS Entries	64
Load/Store Queue Entries	48/48
Dependence Predictor	4096-entry Bloom Filter

Caches and Memory	
L1 Cache Size	32 KB instruction, 32 KB data
L1 Line Width	64 bytes
L1 Associativity	8
L1 Hit Latency	3 cycles
ITLB/DTLB Entries	128/256
L2 Cache Size	2 MB
L2 Line Width	64 bytes
L2 Associativity	8
L2 Hit Latency	12
Memory Latency	50 ns (104 cycles)

NPU	
Number of PEs	8
Bus Schedule FIFO	512×20-bit
Input FIFO	128×32-bit
Output FIFO	128×32-bit
Config FIFO	8×32-bit

NPU PE	
Weight Cache	512×33-bit
Input FIFO	8×32-bit
Output Register File	8×32-bit
Sigmoid Unit LUT	2048×32-bit
Multiply-Add Unit	32-bit Single-Precision FP

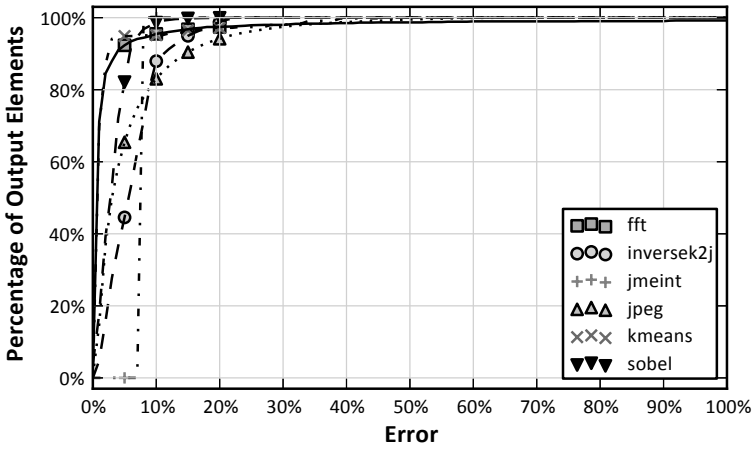


Figure 5.6: Cumulative distribution function (CDF) of the applications' output error. A point (x, y) indicates that y fraction of the output elements see error $\leq x$.

The Parrot transformation degrades each application's average output quality by less than 10%, a rate commensurate with other approximate computing techniques.

To study the application level quality loss in more detail, Figure 5.6 depicts the CDF (cumulative distribution function) plot of final error for each element of application's output. The output of each benchmark consists of a collection of elements—an image consists of pixels; a vector consists of scalars; etc. The error CDF reveals the distribution of output errors among an application's output elements and shows that very few output elements see large quality loss.

The majority (80% to 100%) of each transformed application's output elements have error less than 10%.

5.7.2 Experimental Setup

Cycle-accurate simulation. We use the MARSSx86 cycle-accurate x86-64 simulator [91] to evaluate the performance effect of the Parrot transformation and NPU acceleration. Table 5.2 summarizes the microarchitectural parameters for the core, memory subsystem, and NPU. We configure the simulator to resemble Intel's Penryn microarchitecture, which is an aggressive out-of-order design. We augment MARSSx86 with a cycle-accurate NPU simulator and add support for NPU queue instructions through unused x86 opcodes. We use C assembly inlining to add the NPU invocation code. We compile the benchmarks using GCC version 4.4.6 with the -O3 flag to enable aggressive

compiler optimizations. The baseline in all of the reported results is the execution of the entire benchmark on the core without the Parrot transformation.

Energy modeling. MARSSx86 generates an event log during the cycle-accurate simulation of the program. The resulting statistics are sent to a modified version of McPAT [77] to estimate the energy consumption of each execution. We model the energy consumption of an 8-PE NPU using the results from McPAT and CACTI 6.5 [86] for memory arrays, buses, and steering logic. We use the results from Galal et al. [48] to estimate the energy of multiply-and-add operations. We model the NPU and the core at the 45 nm technology node. The NPU operates at the same frequency and voltage as the main core. We use the 2080 MHz frequency and $V_{dd} = 0.9\ V$ settings because the energy results in Galal et al. [48] are for this frequency and voltage setting.

5.7.3 Experimental Results

Dynamic instruction subsumption. Figure 5.7 depicts dynamic instruction count of each transformed benchmark normalized to the instruction count for CPU-only execution. The figure divides each application into NPU communication instructions and application instructions. While the potential benefit of NPU acceleration is directly related to the amount of CPU work that can be elided, the queuing instructions and the cost of neural network evaluation limit the actual benefit. For example, inversek2j exhibits the greatest potential for benefit: even accounting for the communication instructions, the transformed program executes 94% fewer instructions on the core. Most of the benchmark's dynamic instructions are in the target region for the Parrot transformation and it only communicates four values with the NPU per invocation. This is because inversek2j is an ideal case: the entire algorithm has a fixed-size input ((x, y) coordinates of the robot arm), fixed-size output ((θ_1, θ_2) angles for the arm joints), and tolerance for imprecision. In contrast, kmeans is representative of applications where the Parrot transformation applies more locally: the target code is "hot" but only consists of a few arithmetic operations and the communication overhead is relatively high.

Performance and energy benefits. Figure 5.8a shows the application speedup when an 8-PE NPU is used to replace each benchmark's target function. The rest of the code runs on the core. The

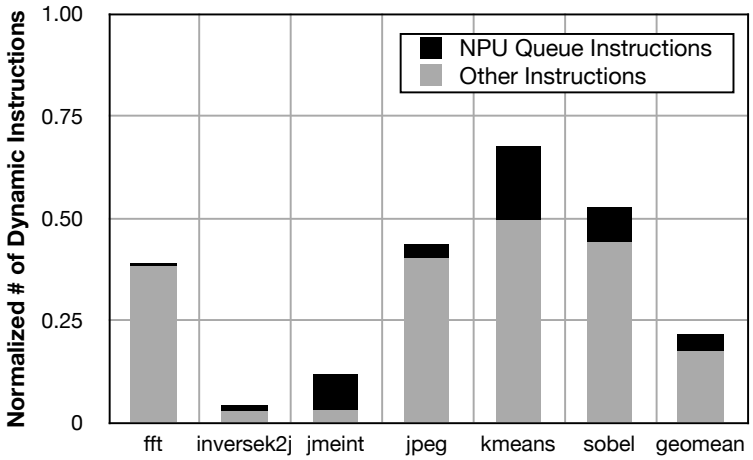
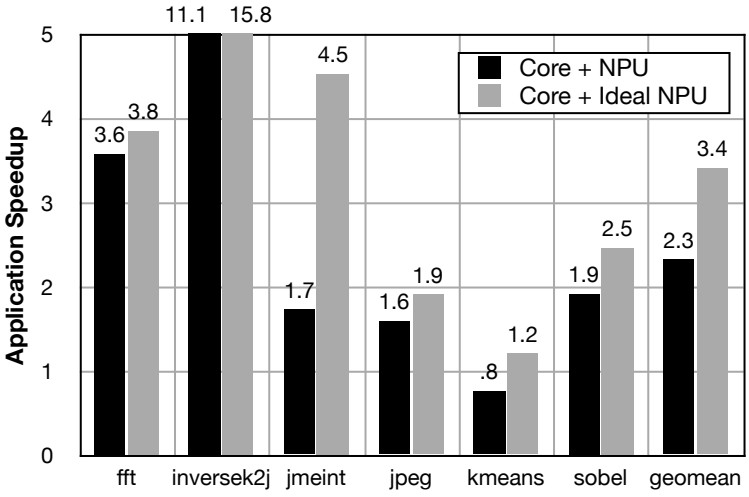


Figure 5.7: Number of dynamic instructions after Parrot transformation normalized to the original program.

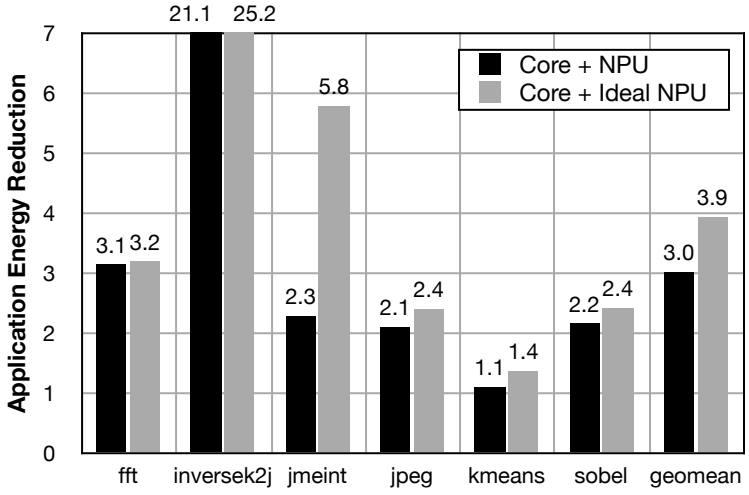
baseline is executing the entire, untransformed benchmark on the CPU. The plots also show the potential available speedup: the hypothetical speedup if the NPU takes zero cycles for computation. Among the benchmarks $\mathrm{inversek2j}$ sees the highest speedup (11.1×) since the Parrot transformation substitutes the bulk of the application with a relatively small NN ($2 \rightarrow 8 \rightarrow 2$). On the other hand, kmeans sees a 24% slowdown even though it shows a potential speedup of 20% in the limit. The transformed region of code in kmeans consists of 26 mostly arithmetic instructions that can efficiently run on the core while the NN ($6 \rightarrow 8 \rightarrow 4 \rightarrow 1$) for this benchmark is comparatively complex and involves more computation (84 multiply-adds and 12 sigmoids) than the original code. On average, the benchmarks see a speedup of 2.3× through NPU acceleration.

Figure 5.8b shows the energy reduction for each benchmark. The baseline is the energy consumed by running the entire benchmark on the unmodified CPU and the ideal energy savings for a hypothetical zero-energy NPU. The Parrot transformation elides the execution of significant portion of dynamic instructions that otherwise would go through power-hungry stages of the OoO pipeline. The reduction in the number of dynamic instructions and the energy-efficient design of the NPU yield a 3.0× average application energy reduction.

For the applications we studied, the Parrot transformation and NPU acceleration provided an average 2.3× speedup and 3.0× energy reduction.



(a) Total application speedup with 8-PE NPU



(b) Total application energy saving with 8-PE NPU

Figure 5.8: Performance and energy improvements.

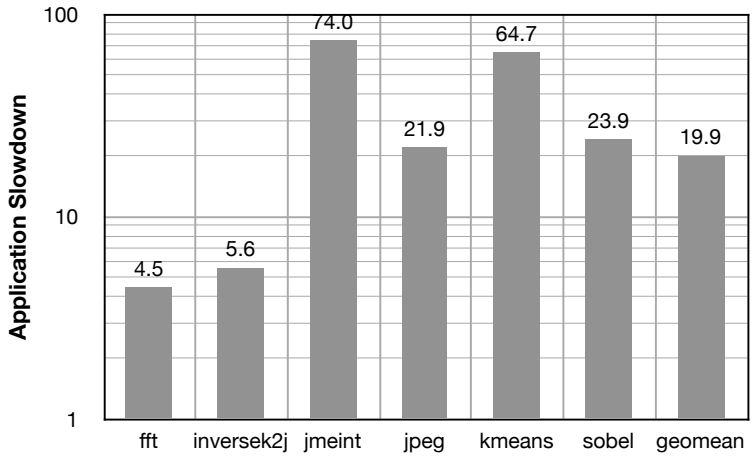


Figure 5.9: Slowdown with software neural network execution.

Results for a hypothetical zero-cost NPU suggest that, in the limit, more efficient implementation techniques such as analog NPUs could result in up to 3.4× performance and 3.7× energy improvements on average.

Software neural network execution. While our design evaluates neural networks on a dedicated hardware unit, it is also possible to run transformed programs entirely on the CPU using a software library for neural network evaluation. To evaluate the performance of this all-software configuration, we executed each transformed benchmark using calls to the widely-used Fast Artificial Neural Network (FANN) library [46] in place of NPU invocations. Figure 5.9 shows the slowdown compared to the baseline (untransformed) execution of each benchmark. Every benchmark exhibits a significant slowdown when the Parrot transformation is used without NPU acceleration. `jmeint` shows the highest slowdown because 1079 x86 instructions—which take an average of 326 cycles on the core—are replaced by 928 multiplies, 928 adds, and 42 sigmoids. FANN's software multiply-and-add operations involve calculating the address of the neuron weights and loading them. The overhead of function calls in the FANN library also contributes to the slowdown.

The Parrot transformation requires efficient neural network execution, such as hardware acceleration, to be beneficial.

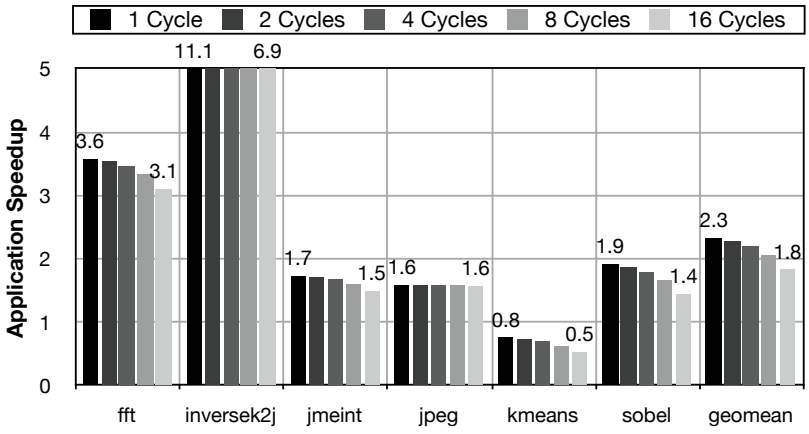


Figure 5.10: Sensitivity of the application's speedup to NPU communication latency. Each bar shows the speedup if communicating with the NPU takes n cycles.

Sensitivity to communication latency. The benefit of NPU-based execution depends on the cost of each NPU invocation. Specifically, the latency of the interconnect between the core and the NPU can affect the potential energy savings and speedup. Figure 5.10 shows the speedup for each benchmark under five different communication latencies. In each configuration, it takes n cycles to send data to the NPU and n cycles to receive data back from the NPU. In effect, $2n$ cycles are added to the NPU invocation latency. We imagine a design with pipelined communication, so individual enqueue and dequeue instructions take one extra cycle each in every configuration.

The effect of communication latency varies depending on the application. In cases like jpeg, where the NPU computation latency is significantly larger than the communication latency, the speedup is mostly unaffected by increased latency. In contrast, inversek2j sees a significant reduction in speedup from 11.1× to 6.9× when the communication latency increases from one cycle to 16 and becomes comparable to the computation latency. For kmeans, the slowdown becomes 48% for a latency of 16 cycles compared to 24% when the communication latency is one cycle.

For some applications with simple neural network topologies, a tightly-coupled, low-latency NPU–CPU integration design is highly beneficial. Other applications we studied can tolerate a higher-latency interconnect.

Number of PEs. Figure 5.11 shows the geometric mean speedup gain from doubling the number of PEs in the NPU. Doubling the number of PEs beyond eight yields less than 5% geometric

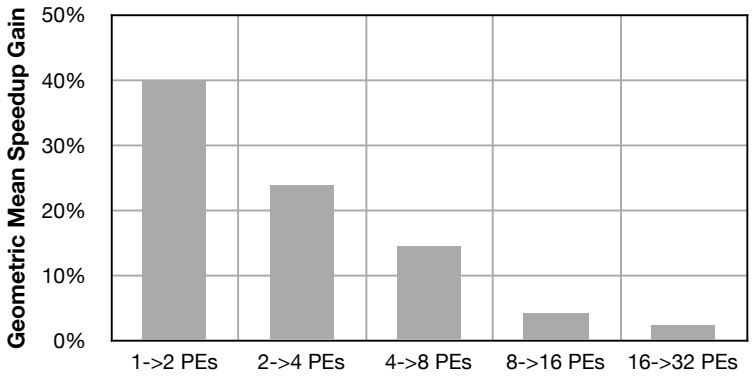


Figure 5.11: Performance gain per doubling the number of PEs.

mean speedup gain, which does not justify the complexity of adding more than eight PEs for our benchmarks.

5.8 Limitations and Future Directions

Our results suggest that the Parrot transformation and NPU acceleration can provide significant performance and energy benefits. However, further research must address three limitations to the Parrot transformation as described in this work: (1) applicability; (2) programmer effort; and (3) quality and error control.

Applicability. Since neural networks inherently produce approximate results, not all code regions can undergo the Parrot transformation. As enumerated in Section 5.3.1, a target code region must satisfy the following conditions:

- The region must be hot in order to benefit from acceleration.
- The region must be approximable. That is, the program must incorporate application-level tolerance of imprecision in the results of the candidate region.
- The region must have a bounded number of statically identifiable inputs and outputs.

Although these criteria form a basis for programmers or compilers to identify nominees for the Parrot transformation, they do not guarantee that the resulting neural network will accurately approximate the code region. There is no simple criterion that makes a certain task (here a candidate region) suited for learning by a neural network. However, our experience and results suggest that

empirical assessment is effective to classify a wide variety of approximate functions as NPU-suitable. Follow-on work can improve on empirical assessment by identifying static code features that tend to indicate suitability for learning-based acceleration.

Programmer effort. In this work, the Parrot transformation requires programmers to (1) identify approximable code regions and (2) provide application inputs to be used for training data collection.

As with the other approaches that ensure the safety of approximate computation and avoid catastrophic failures [98], the programmer must explicitly provide information for the compiler to determine which code regions are safe to approximate. As Section 5.3.2 outlines, future work should explore allowing the compiler to automatically infer which blocks are amenable to approximation.

Because NPU acceleration depends on representative test cases, it resembles a large body of other techniques that use programmer-provided test inputs, including quality assurance (e.g., unit and integration testing) and profile-driven compilers. Future work should apply traditional coverage measurement and improvement techniques, such as test generation, to the Parrot transformation. In general, however, we found that it was straightforward to provide sufficient inputs for the programs we examined. This is in part because the candidate function is executed many times in a single application run, so a small number of inputs can suffice. Furthermore, as Section 5.4.2 mentions, an on-line version of the Parrot transformation workflow could use samples of post-deployment inputs if representative tests are not available pre-deployment.

Quality and error control. The results in this chapter suggest that NPU acceleration can effectively approximate code with accuracy that is commensurate with state-of-the art approximate computing techniques. However, there is always a possibility that, for some inputs, the NPU computes a significantly lower-quality result than the average case. In other words, without exhaustively exploring the NPU's input space, it is impossible to give guarantees about its worst-case accuracy.

This unpredictability is common to other approximation techniques [98, 40]. As long as the frequency of low-quality results is low and the application can tolerate these infrequent large errors, approximation techniques like NPUs can be effective. For this reason, future research should explore mechanisms to mitigate the frequency of such low-quality results. One such mechanism

is to predict whether the NPU execution of the candidate region will be acceptable. For example, one embodiment would check whether an input falls in the range of inputs seen previously during training. If the prediction is negative, the original code can be invoked instead of the NPU. Alternatively, the runtime system could occasionally measure the error by comparing the NPU output to the original function's output. In case the sampled error is greater than a threshold, the neural network can be retrained. These techniques are similar in spirit to related research on estimating error bounds for neural networks [110].

5.9 Concluding Remarks

This chapter demonstrated that neural accelerators can successfully mimic diverse regions of approximable imperative code. The Parrot algorithmic transformation converts different regions of code to a common neural-network representation. Using neural networks as the common representation enables a new class of accelerators, neural processing units (NPUs), that yield significant application-level energy and performance savings. The levels of error introduced are comparable to those seen in previous approximate computing techniques. This work leads to the following two key insights. First, the program transformation must consider a range of neural network topologies; a single topology is ineffective across diverse applications. Second, the accelerator must be tightly coupled with a processor's pipeline to accelerate fine-grained regions of code. By providing an end-to-end solutions to meet these key requirements, the evaluated application suite ran 2.3× faster on average while using 3.0× less energy and maintaining accuracy greater than 90% in all cases.

Traditionally, hardware implementations of neural networks have been confined to specific classes of learning applications. In this work, we show that the potential exists to use neural hardware to accelerate general-purpose code that can tolerate small errors. This acceleration capability aligns with both transistor and application trends, as transistors become less reliable and as imprecise applications grow in importance. NPUs form a new class of trainable accelerators with potential implementations in the digital and analog domains.

Chapter 6

Related Work

This dissertation represents a convergence of seven main bodies of research: power-performance measurement, modeling single-core and multicore processors, approximate computing, voltage overscaling, information flow tracking, general-purpose configurable accelerators, and neural networks. In this chapter, we briefly overview the related work on each of these bodies of work.

6.1 Power-Performance Measurement

The processor design literature is now full of *performance* measurement and analysis. a tradition that began in the 1980s [29]. Despite the growing importance of power, power measurements are still relatively rare [44, 66, 73]. Here we only summarize related power measurement and simulation work.

Power measurement. Isci and Martonosi combine a clamp ammeter with performance counters for per unit power estimation of the Intel Pentium 4 on SPEC CPU2000 [66]. Bircher and John Bircher and John study power using a series resistor, sampling the voltage across the resistor at 1KHz on the AMD quad core Opteron and Phenom processors [8]. Fan et al. estimate whole system power for large scale data centers [44]. They find that even the most power-consuming workloads draw less than 60% of peak possible power consumption. We measure chip power, and support their results by showing that TDP does not predict measured chip power. Our work is the first to compare microarchitectures, technology generations, individual benchmarks, and workloads in the context of power and performance.

Power modeling. Power modeling is necessary to thoroughly explore architecture design [3, 79, 77]. Measurement complements simulation by providing validation. For example, some prior simulators used TDP, but our measurements show it is not accurate. As we look to the future, we believe programmers will need to tune their applications for power and energy, not only performance. Just as hardware performance counters provide insight to applications, so will power and energy measurements.

Methodology. Although the results show conclusively that managed and native workloads have different responses to architectural variations, perhaps this result should not be surprising. Unfortunately, few architecture or operating system publications with processor measurements or simulated designs use Java or any other managed workloads, even though the evaluation methodologies we use here for real processors and those for simulators are well developed [9, 55, 49]

6.2 Modeling Multicores

Hill and Marty extend Amdahl's Law to model multicore speedup with symmetric, asymmetric, and dynamic topologies and conclude dynamic multicores are superior [62]. Several extensions to Hill and Marty model have been developed for modeling 'uncore' components (e.g. interconnection network and last level cache), [81], computing core configuration optimal for energy [74, 18], and leakage power [115]. All these model uses area as the primary constraint and model single-core area/performance tradeoff using Pollack's rule (Performance $\propto \sqrt{Area}$ [92]) without considering technology trends.

Azizi et al. derive the single-core energy/performance trade-off as Pareto frontiers using architecture-level statistical models combined with circuit-level energy-performance trade-off functions [3]. For modeling single-core power/performance and area/performance trade-offs, our core model derives two separate Pareto frontiers from empirical data. Further, we project these trade-off functions to the future technology nodes using our device model. We perform a power/energy Pareto efficiency analysis at 45 nm using total chip power measurements in the context of a retrospective workload and microarchitecture analysis [33]. In contrast to the total chip power measurements for specific workloads, we use the power and area budget allocated to a single-core to derive the

Pareto frontiers and combine those with our device and chip-level models to study the future of multicore design and the implications of technology scaling.

Chakraborty considers device-scaling and estimates a simultaneous activity factor for technology nodes down to 32 nm [14]. Hempstead et al. introduce a variant of Amdahl's Law to estimate the amount of specialization required to maintain 1.5× performance growth per year, assuming completely parallelizable code [61]. Chung et al. study unconventional cores including custom logic, FPGAs, or GPUs in heterogeneous single-chip design [19]. They rely on Pollack's rule for the area/performance and power/performance tradeoffs. Using ITRS projections, they report on the potential for unconventional cores considering parallel kernels. Hardavellas et al. forecast the limits of multicore scaling and the emergence of dark silicon in servers with workloads that have an inherent abundance of parallelism [57]. Using ITRS projections, Venkatesh et al. estimate technology-imposed utilization limits and motivate energy-efficient and application-specific core designs [112].

Previous work largely abstracts away processor organization and application details. This study provides a comprehensive model that considers the implications of process technology scaling, decouples power/area constraints, uses real measurements to model single-core design trade-offs, and exhaustively considers multicore organizations, microarchitectural features, and real applications and their behavior.

6.3 Approximate Computing

Many categories of "soft" applications have been shown to be tolerant to imprecision during execution [45, 114, 78, 24]. These studies show that there are significant body of application that can tolerate inaccuracy in computation.

6.3.1 Approximation in Software

Previous work has exposed relaxed semantics in the programming language to give programmers control over the precision of software [101, 98, 4].

Loop perforation [101] allows programmers to systematically skips iterations of the loops and trade accuracy for performance. Green [4] provides a framework for programmers to specify which regions of code can be approximated. EnerJ [98] uses type qualifiers to identify approximate and

precise slices of the code. As an implementation of approximate semantics, both the truffle microarchitecture and the Parrot transformation dovetails with these programming models.

6.3.2 Approximation in Hardware

Prior work has also explored relaxed hardware semantics and their impact on these applications, both (1) in the form of fully approximate units of computation [15, 60, 76, 87], and (2) as extensions to traditional architectures [1, 25, 40, 80].

Fully approximate units of computation. A significant amount of prior work has proposed hardware that compromises on execution correctness for benefits in performance, energy consumption, and yield. ERSA proposes collaboration between discrete reliable and unreliable cores for executing error-resilient applications [76]. Stochastic processors encapsulate another proposal for variable-accuracy functional units [87]. Probabilistic CMOS (PCMOS) proposes to use the probability of low-voltage transistor switching as a source of randomness for special randomized algorithms [15]. Finally, algorithmic noise-tolerance (ANT) proposes approximation in the context of digital signal processing [60]. Imprecise integer logic blocks have also been designed for bio-inspired soft algorithms [82]. These proposals [76, 15, 87, 82] do not provide any generalizable approach, programming model, or detailed processor microarchitecture for approximation.

Our proposed dual-voltage design, Truffle, in contrast, supports fine-grained, single-core approximation that leverages language support for explicit approximation in general-purpose applications. It does not require manual offloading of code to coprocessors and permits fully-precise execution on the same core as low-power approximate instructions. Truffle extends general-purpose CPUs; it is not a special-purpose coprocessor.

In these proposals, the entire unit of computation carries relaxed semantics and thus require developing vastly different programming models. In contrast, NPUs can be used with conventional imperative languages and existing code. No special code must be written to take advantage of the approximate unit; only lightweight annotation is required.

6.3.3 Extensions to traditional architectures.

The second category of techniques have explored relaxed hardware semantics that extend traditional architectures. Relax is a compiler/architecture system for suppressing hardware fault recovery in certain regions of code, exposing these errors to the application [25]. Fuzzy memoization stores the values of floating point operations in a lookup table and replaces executing floating point instructions with lookup, when the operands are close enough to the stored operands [1]. Bit-width reduction [109] is another orthogonal-to-voltage-overscaling technique for approximating floating-point operations.

A Truffle-like architecture supports approximation at a single-instruction granularity, exposes approximation in storage elements, and guarantees precise control flow even when executing approximate code. In addition, Truffle goes further and elides fault *detection* as well as recovery where it is not needed. Broadly, the key difference between Truffle and prior work is that Truffle was co-designed with language support. Specifically, relying on disciplined approximation with strong static guarantees offered by the compiler and language features enables an efficient and simple design. Static guarantees also lead to strong safety properties that significantly improve programmability.

Since all the instructions, both approximate and precise, still run on the core, the benefits of approximation are limited for this class of techniques. In addition, these techniques's fine granularity precludes higher-level, algorithmic transformations that take advantage of approximation. The Parrot transformation operates at coarser granularities—from small functions to entire algorithms—and potentially increases the benefits of approximation. Furthermore, NPU acceleration reduces the number of instructions that go through the power-hungry frontend stages of the processor pipeline.

6.4 Voltage Overscaling

Previous work has also explored dual-V_{dd} designs for power optimization in fully-precise computers [117, 16, 93]. Truffle's instruction-controlled voltage changes make it fundamentally different from these previous techniques. Razor and related techniques also use voltage underscaling for energy reduction but use error recovery to hide errors from the application [30, 69]. Disciplined general-

purpose approximate computation can enable energy savings beyond those allowed by correctness-preserving optimizations.

6.5 Information Flow Tracking

Truffle resembles architectures that incorporate information flow tracking for security [108, 104, 22]. In that work, the hardware enforces information flow invariants dynamically based on tags provided by the application or operating system. With Truffle, the *compiler* provides the information flow invariant, freeing the architecture from costly dynamic checking.

6.6 General-Purpose Configurable Accelerators

The Parrot transformation and NPU acceleration extends prior work on configurable computing, synthesis, specialization, and acceleration that focuses on compiling traditional, imperative code for efficient hardware structures. One research direction seeks to synthesize efficient circuits or configure FPGAs to accelerate general-purpose code [95, 96, 20, 43]. Similarly, static specialization has shown significant efficiency gains for irregular and legacy code [112, 113]. More recently, configurable accelerators have been proposed that allow the main CPU to offload certain code to a small, efficient structure [50, 52]. These techniques, like NPU acceleration, typically rely on profiling to identify frequently executed code sections and include compilation workflows that offload this "hot" code to the accelerator. This work differs in its focus on accelerating *approximate* code. NPUs represent an opportunity to go beyond the efficiency gains that are possible when strict correctness is not required. While some code is not amenable to approximation and should be accelerated only with correctness-preserving techniques, NPUs can provide greater performance and energy improvements in many situations where relaxed semantics are appropriate.

6.7 Neural Networks

There is an extensive body of work on hardware implementation of neural networks (neural hardware) both digital [94, 31, 119] and analog [13, 100, 106, 68]. Recent work has proposed higher-level abstractions for implementation of neural networks [59]. Other work has examined fault-tolerant hardware neural networks [58, 107]. In particular, Temam [107] uses datasets from the

UCI machine learning repository [47] to explore fault tolerance of a hardware neural network design. That work suggests that even faulty hardware can be used for efficient simulation of neural networks. The Parrot algorithmic transformation provides a compiler workflow that allows general-purpose approximate applications to take advantage of this and other hardware neural networks.

While we use neural networks to approximate regions of code written in conventional programming languages, Grzeszczuk et al. use neural networks to learn and emulate the physical movement of object is animation [51]. The neural emulation provides significant speedup even without hardware support. This is a new domain of applications that NPU can be significantly beneficial.

An early version of this work [39] proposed the core idea of automatically mapping approximable regions of imperative code to neural networks. A more recent study [17] showed that 5 of 13 applications from the PARSEC suite can be manually reimplemented to make use of various kinds of neural networks, demonstrating that some applications allow higher-level algorithmic modifications to make use of hardware neural networks (and potentially an architecture like NPUs). However, that work did not prescribe a programming model, a compilation workflow, nor a preferred hardware architecture.

Fundamentally, the Parrot transformation leverages hardware neural networks to create a new class of configurable accelerators for approximate programs.

CHAPTER 7

A PATH FORWARD

For decades, Moore's Law *plus* Dennard scaling permitted more transistors, faster transistors, and more energy efficient transistors with each new process node, justifying the enormous costs required to develop each new process node. Dennard scaling's failure led industry to race down the multicore path, which for some time permitted performance scaling for parallel and multitasked workloads, allowing the economics of process scaling to hold. A key question for the computing community is whether scaling multicores will provide the performance and value needed to scale down many more technology generations. Are we in a long-term multicore "era," or it instead will be a "multicore decade" (2004-2014)? Will industry need to move in different, perhaps radical, directions to justify the cost of scaling? To answer the question, our dark silicon study modeled an upper bound on parallel application performance available from multicore and CMOS scaling–assuming no major disruptions in process scaling or core efficiency. Using a constant area and power budget, this study showed that the space of known multicore designs (CPU, GPU, their hybrids) or novel heterogeneous topologies (e.g., dynamic or composable) falls far short of the historical performance gains to which the microprocessor industry is accustomed. Even with aggressive ITRS scaling projections, scaling cores achieves a geometric mean 7.9$\times$ speedup in ten years at 8 nm–a 23% annual gain. Our findings suggest that without process breakthroughs, directions beyond multicore are needed to provide performance scaling. There are reasons to be both optimistic and pessimistic.

The optimistic view. Our dark silicon study shows if energy efficiency breakthroughs are made on supply voltage and process scaling, the performance improvement potential for multicore scaling is still high for applications with very high degrees of parallelism.

The pessimists view. A pessimistic interpretation of our dark silicon study is that the performance improvements to which we have grown accustomed over the past 40 years are unlikely to continue with multicore scaling as the primary driver. The transition from multicore to a new approach is likely to be more disruptive than the transition to multicore. Furthermore, to sustain the current cadence of Moore's Law, the transition needs to be made in only a few years, much shorter than the traditional academic time frame for research and technology transfer. Major architecture breakthroughs in "alternative" directions such as neuromorphic computing, quantum computing, or bio-integration will require even more time to enter the industrial product cycle. Furthermore, while a slowing of Moore's Law will obviously not be fatal, it has significant economic implications for the semiconductor industry.

Efficiency through specialization. Recent work has quantified three orders of magnitude difference in efficiency between general-purpose processors and ASICs [56]. However, there is a well-known tension between efficiency and programmability. Designing ASICs for the massive base of quickly changing general-purpose applications is currently infeasible. Programmable accelerators, such as GPUs and FPGAs, and specialized hardware can provide an intermediate point between the efficiency of ASICs and the generality of conventional processors, gaining significant improvements for specific domains of applications. Even though there is an emerging consensus that specialization and acceleration is a promising approach for efficiently utilizing the growing number of transistors, developing programming abstractions that allow general-purpose applications to leverage specialized hardware and programmable accelerators remains challenging.

The need for microarchitecture innovations. Our dark silicon study shows that fundamental processing limitations emanate from the processor core. The limited improvements on single-threaded performance is the inhibiting factor. Clearly, architectures that move well past the power/performance Pareto-optimal frontier of today's designs are necessary to bridge the dark silicon gap and

utilize the increases in transistor count. Hence, improvements to the processor core efficiency will have significant impact on performance improvement and will enable technology scaling even though the core consumes only 20% of the power budget for an entire laptop, smartphone, tablet, etc. When performance becomes limited, microarchitectural techniques that occasionally use parts of the chip to deliver outcomes orthogonal to performance, such as security, programmer productivity, and software maintainability are ways to sustain the economics of the industry. We believe our dark silicon study will revitalize and trigger microarchitecture innovations, making the case for their urgency and their potential impact.

Opportunity for disruptive innovations. Our study is based on a model that takes into account properties of devices, processor cores, multicore organizations and topologies. Thus the model inherently provides the places to focus on for innovation. To surpass the dark silicon performance barrier highlighted by our work, designers must develop systems that use significantly more energy-efficient techniques. Some examples include device abstractions beyond digital logic (error-prone devices); processing paradigms beyond superscalar, SIMD, and SIMT; and program semantic abstractions allowing probabilistic and approximate computation. There is en emerging synergy between the applications that can tolerate approximation and the unreliability in the computation fabric as technology scales down. If done in a disciplined manner, relaxing the high tax of providing perfect accuracy at the device, circuit, and architecture level can provide a huge opportunity to improve performance and energy efficiency for the domains in which applications can tolerate approximate computation yet deliver acceptable outputs. The results from our dark silicon study show that such radical departures are necessary and our model provides quantitative measures to examine the impact of such techniques.

General-purpose approximate computing. The work in this dissertation showed that not only it is possible to relax the abstraction of full-accuracy in general-purpose computing without breaking the safety guarantees, but also significant gains both in performance and efficiency are achievable. This dissertation showed that adding the dimension of error to the design space of general-purpose processors provides a fertile ground for innovation. We explored this three-dimensional space and designed approximate von Neumann architectures that can trade accuracy for efficiency at the level

of single instructions. Even thought the efficiency gains are significant—up to43%, they are limited due to the limited applicability of the approximation techniques in von Neumann architectures. The overhead of fetching, decoding, and committing instructions that need to be precise at all times curtails the benefits from approximation.

However, by relaxing the abstraction of full accuracy, we showed that it is *possible* to convert regions of code from a von Neumann model of computing to a neural mode of computing. We proposed an algorithmic transformation that automatically selects and trains a neural network to mimic a region of code, developed in conventional programming languages. One of the most important findings of our work is that neural networks can learn the behavior of regions of code. After the learning phase, the compiler transparently replaces the original code with an invocation of a low-power accelerator. The transformation is beneficial because neural networks are parallel structures with efficient hardware implementations. Leveraging this transformation, we introduced a new class of accelerators, called Neural Processing Units (NPUs), with implementation potential in both the digital and the analog domain. Even though hardware neural networks are far from being a new idea, prior research has not considered tightly integrating neural hardware with the processor due to the lack of abstractions allowing applications to benefit from the integration. Our work enables neural hardware to operate beyond their conventional use case and accelerate code written in conventional programming languages. This work firmly shows that there are opportunities for approximate computing *beyond* the conventional approaches of computing.

We show that there is a potential new space of algorithmic transformations that leverage learning to mimic regions of code. This space of learning-based transformation can enable the use of alternative technologies for general-purpose computing. For example, using analog circuits to build NPUs provides an opportunity to build hybrid analog/digital general-purpose processors. The use of analog and alternative storage technologies such as memristors in our framework is another direction that is enabled by the work in this dissertation. This new class of accelerators, NPUs, shows that significant performance and efficiency gains are possible when the abstraction of full accuracy is relaxed in general-purpose computing.

Even though dark silicon poses a grand challenge to the entire computing industry, there is a silver lining for architects. The onus is on computer architects—and computer architects only—to deliver performance and efficiency gains that can work across a wide range of problems and keep

the economy of our industry, *the industry of new possibilities*, vibrant and thriving. In general when conventional approaches run out of steam, it is time for extreme creativity. It promises to be an exciting time.

Bibliography

[1] Carlos Alvarez, Jesus Corbal, and Mateo Valero. Fuzzy memoization for floating-point multimedia applications. *IEEE Trans. Comput.*, 54(7), 2005.

[2] Gene M. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS*, 1967.

[3] Omid Azizi, Aqeel Mahesri, Benjamin C. Lee, Sanjay J. Patel, and Mark Horowitz. Energy-performance tradeoffs in processor architecture and circuit design: a marginal cost analysis. In *ISCA*, 2010.

[4] Woongki Baek and Trishul M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *PLDI*, 2010.

[5] 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.

[6] M. Bhadauria, V.M. Weaver, and S.A. McKee. Understanding PARSEC performance on contemporary CMPs. In *IISWC*, 2009.

[7] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, pages 451–460, 2008.

[8] W. Lloyd Bircher and Lizy K. John. Analysis of dynamic power management on multi-core processors. In *ICS*, pages 327–338, 2008.

[9] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffman, A. M. Khan, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. Wake up and smell the coffee: Evaluation methodologies for the 21st century. *Communications of the ACM*, 51(8):83–89, August 2008.

[10] Mark Bohr. A 30 year retrospective on Dennard's MOSFET scaling paper. *IEEE SSCS Newsletter*, pages 11–13, Winter 2007.

[11] Shekhar Borkar. The exascale challenge. Keynote at VLSI-DAT, 2010.

[12] Shekhar Borkar and Andrew A. Chen. The future of microprocessors. *Communications of the ACM*, 54(5), 2011.

[13] Bernhard E. Boser, Eduard Säckinger, Jane Bromley, Yann Lecun, Lawrence D. Jackel, and Senior Member. An analog neural network processor with programmable topology. *J. Solid-State Circuits*, 26:2017–2025, 1991.

[14] Koushik Chakraborty. *Over-provisioned Multicore Systems*. PhD thesis, University of Wisconsin-Madison, 2008.

[15] Lakshmi N. Chakrapani, Bilge E. S. Akgul, Suresh Cheemalavagu, Pinar Korkmaz, Krishna V. Palem, and Balasubramanian Seshasayee. Ultra-efficient (embedded) SOC architectures based on probabilistic CMOS (PCMOS) technology. In *DATE*, 2006.

[16] Chunhong Chen, Ankur Srivastava, and Majid Sarrafzadeh. On gate level power optimization using dual-supply voltages. *IEEE Trans. VLSI Syst.*, 9, 2001.

[17] Tianshi Chen, Yunji Chen, Marc Duranton, Qi Guo, Atif Hashmi, Mikko Lipasti, Andrew Nere, Shi Qiu, Michele Sebag, and Olivier Temam. Benchnn: On the broad potential application scope of hardware neural network accelerators? In *IISWC*, November 2012.

[18] Sangyeun Cho and Rami Melhem. Corollaries to Amdahl's law for energy. *Computer Architecture Letters*, 7(1), January 2008.

[19] Eric S. Chung, Peter A. Milder, James C. Hoe, and Ken Mai. Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPUs? In *MICRO*, pages 225–236, 2010.

[20] Nathan Clark, Manjunath Kudlur, Hyunchul Park, Scott Mahlke, and Krisztian Flautner. Application-specific processing on a general-purpose core via transparent instruction set customization. In *MICRO*, 2004.

[21] Digital Equipment Corporation. *Alpha Architecture Handbook, Version 3*. Digital Equipment Corporation, 1996.

[22] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: A flexible information flow architecture for software security. In *ISCA*, 2007.

[23] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. RAPL: memory power estimation and capping. In *ISLPED*, pages 189–194, 2010.

[24] M. de Kruijf and K. Sankaralingam. Exploring the synergy of emerging workloads and silicon reliability trends. In *SELSE*, 2009.

[25] Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. Relax: An architectural framework for software recovery of hardware faults. In *ISCA*, 2010.

[26] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9, October 1974.

[27] Robert H. Dennard, Jin Cai, and Arvind Kumar. A perspective on today's scaling challenges and possible future directions. *Solid-State Electronics*, 5(4), April 2007.

[28] Chris Edwards. Scary dark silicon is here today, 2009. URL http://blog.shrinkingviolence.com/2009/10/scary-dark-silicon-is-here-tod.html.

[29] Joel S. Emer and Douglas W. Clark. A characterization of processor performance in the VAX-11/780. In *ISCA*, pages 301–310, 1984.

[30] D. Ernst, Nam Sung Kim, S. Das, S. Pant, R. Rao, Toan Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *MICRO*, 2003.

[31] H. Esmaeilzadeh, P. Saeedi, B.N. Araabi, C. Lucas, and S.M. Fakhraie. Neural network stream processing core (NnSP) for embedded systems. In *ISCAS*, 2006.

[32] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *ISCA*, 2011.

[33] Hadi Esmaeilzadeh, Ting Cao, Yang Xi, Stephen M. Blackburn, and Kathryn S. McKinley. Looking back on the language and hardware revolutions: measured power, performance, and scaling. In *ASPLOS*, pages 319–332, 2011.

[34] Hadi Esmaeilzadeh, Ting Cao, Xi Yang, Stephen M Blackburn, and Kathryn S McKinley. Source materials in ACM Digital Library for: Looking back on the language and hardware revolutions: Measured power, performance, and scaling. In *ASPLOS*, pages 319–332, March 2011. URL http://doi.acm.org/10.1145/1950365.1950402.

[35] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Power limitations and dark silicon challenge the future of multicore. *ACM Trans. Comput. Syst.*, 30(3):11:1–11:27, August 2012.

[36] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. *IEEE Micro*, 32(3):122–134, May 2012.

[37] Hadi Esmaeilzadeh, Ting Cao, Xi Yang, Stephen Blackburn, and Kathryn McKinley. What is happening to power, performance, and software? *IEEE Micro*, 32(3):110–121, May 2012.

[38] Hadi Esmaeilzadeh, Ting Cao, Xi Yang, Stephen M. Blackburn, and Kathryn S. McKinley. Looking back and looking forward: power, performance, and upheaval. *Commun. ACM*, 55(7): 105–114, July 2012.

[39] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Towards neural acceleration for general-purpose approximate computing. In *WEED*, June 2012.

[40] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Architecture support for disciplined approximate programming. In *ASPLOS*, 2012.

[41] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Power challenges may end the multicore era. *Commun. ACM*, 56(2):93–102, February 2013.

[42] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. *Micro, IEEE*, 33(3):16–27, 2013.

[43] K. Fan, M. Kudlur, G. Dasika, and S. Mahlke. Bridging the computation gap between programmable processors and hardwired accelerators. In *HPCA*, 2009.

[44] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. In *ISCA*, pages 13–23, 2007.

[45] Yuntan Fang, Huawei Li, and Xiaowei Li. A fault criticality evaluation framework of digital systems for error tolerant video applications. In *ATS*, 2011.

[46] FANN. Fast artificial neural network library, 2012. URL http://leenissen.dk/fann/wp/.

[47] A. Frank and A. Asuncion. UCI machine learning repository, 2010. URL http://archive.ics.uci.edu/ml.

[48] S Galal and M Horowitz. Energy-efficient floating-point unit design. *IEEE Trans. Comput.*, 60 (7):913–922, 2011.

[49] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *OOPSLA*, pages 57–76, 2007.

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

[51] Radek Grzeszczuk, Demetri Terzopoulos, and Geoffrey Hinton. Neuroanimator: fast neural network emulation and control of physics-based models. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '98, pages 9–20, 1998.

[52] 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.

[53] Zvika Guz, Evgeny Bolotin, Idit Keidar, Avinoam Kolodny, Avi Mendelson, and Uri C. Weiser. Many-core vs. many-thread machines: Stay away from the valley. *IEEE Computer Architecture Letters*, 8, January 2009.

[54] Alexander Guzhva, Sergey Dolenko, and Igor Persiantsev. Multifold acceleration of neural network computations using GPU. In *ICANN*, 2009.

[55] J. Ha, M. Gustafsson, S. M. Blackburn, and K. S. McKinley. Microarchitectural Characterization of Production JVMs and Java Workloads. In *IBM CAS Workshop*, February 2008.

[56] 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.

[57] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4):6–15, July–Aug. 2011.

[58] Atif Hashmi, Hugues Berry, Olivier Temam, and Mikko H. Lipasti. Automatic abstraction and fault tolerance in cortical microarchitectures. In *ISCA*, 2011.

[59] Atif Hashmi, Andrew Nere, James Jamal Thomas, and Mikko Lipasti. A case for neuromorphic ISAs. In *ASPLOS*, 2011.

[60] Rajamohana Hegde and Naresh R. Shanbhag. Energy-efficient signal processing via algorithmic noise-tolerance. In *ISLPED*, 1999.

[61] Mark Hempstead, Gu-Yeon Wei, and David Brooks. Navigo: An early-stage model to study power-contrained architectures and specialization. In *MoBS*, 2009.

[62] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7), July 2008.

[63] J. Howard et al. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *ISSCC*, 2010.

[64] M. S. Hrishikesh, Doug Burger, Norman P. Jouppi, Stephen W. Keckler, Keith I. Farkas, and Premkishore Shivakumar. The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays. In *International Symposium on Computer Architecture*, pages 14–24, 2002.

[65] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. Core fusion: accommodating software diversity in chip multiprocessors. In *ISCA*, pages 186–197, 2007.

[66] Canturk Isci and Margaret Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *MICRO*, pages 93–104, December 2003.

[67] ITRS. International technology roadmap for semiconductors, 2010 update, 2011. URL http://www.itrs.net.

[68] Antoine Joubert, Bilel Belhadj, Olivier Temam, and Rodolphe Heliot. Hardware spiking neurons design: Analog or digital? In *IJCNN*, 2012.

[69] A.B. Kahng, Seokhyeong Kang, R. Kumar, and J. Sartori. Designing a processor from the ground up to allow voltage/reliability tradeoffs. In *HPCA*, 2010.

[70] R. E. Kessler. The Alpha 21264 Microprocessor. *MICRO*, 1999.

[71] Changkyu Kim, Simha Sethumadhavan, M. S. Govindan, Nitya Ranganathan, Divya Gulati, Doug Burger, and Stephen W. Keckler. Composable lightweight processors. In *MICRO*, 2007.

[72] Wonyoung Kim, David Brooks, and Gu-Yeon Wei. A fully-integrated 3-level DC/DC converter for nanosecond-scale DVS with fast shunt regulation. In *ISSCC*, 2011.

[73] Etienne Le Sueur and Gernot Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Workshop on Power Aware Computing and Systems*, Vancouver, Canada, October 2010.

[74] Jeong-Gun Lee, Eungu Jung, and Wook Shin. An asymptotic performance/energy analysis and optimization of multi-core architectures. In *ICDCN*, 2009.

[75] Victor W Lee et al. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *ISCA*, 2010.

[76] Larkhoon Leem, Hyungmin Cho, Jason Bau, Quinn A. Jacobson, and Subhasish Mitra. ERSA: Error resilient system architecture for probabilistic applications. In *DATE*, 2010.

[77] 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.

[78] Xuanhua Li and Donald Yeung. Exploiting soft computing for increased fault tolerance. In *ASGI*, 2006.

[79] Yingmin Li, Benjamin Lee, David Brooks, Zhigang Hu, and Kevin Skadron. CMP design space exploration subject to physical contraints. In *HPCA*, pages 17–28, Feb 2006.

[80] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. Flikker: Saving refresh-power in mobile devices through critical data partitioning. In *ASPLOS*, 2011.

[81] Gabriel Loh. The cost of uncore in throughput-oriented many-core processors. In *ALTA*, 2008.

[82] H. R. Mahdiani, A. Ahmadi, S. M. Fakhraie, and C. Lucas. Bio-inspired imprecise computational blocks for efficient VLSI implementation of soft-computing applications. *Trans. Cir. Sys. Part I*, 57, 2010.

[83] Sasa Misailovic, Stelios Sidiroglou, Hank Hoffman, and Martin Rinard. Quality of service profiling. In *ICSE*, 2010.

[84] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[85] Trevor Mudge. Power: A first-class architectural design constraint. *Computer*, 34(4):52–58, April 2001.

[86] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *MICRO*, 2007.

[87] Sriram Narayanan, John Sartori, Rakesh Kumar, and Douglas L. Jones. Scalable stochastic processors. In *DATE*, 2010.

[88] NetBSD Documentation. How lazy FPU context switch works, 2011. URL http://www.netbsd.org/docs/kernel/lazyfpu.html.

[89] Koichi Nose and Takayasu Sakurai. Optimization of VDD and VTH for low-power and high speed applications. In *ASP-DAC*, 2000.

[90] Kyoung-Su Oh and Keechul Jung. GPU implementation of neural networks. *Pattern Recognition*, 37(6):1311–1314, 2004.

[91] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. MARSSx86: A full system simulator for x86 CPUs. In *DAC*, 2011.

[92] F.J. Pollack. New microarchitecture challenges in the coming generations of CMOS process technologies. Keynote at MICRO, 1999.

[93] Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and T. N. Vijaykumar. Gated-vdd: a circuit technique to reduce leakage in deep-submicron cache memories. In *ISLPED*, pages 90–95, 2000.

[94] K.W. Przytula and V.K. Prasanna Kumar, editors. *Parallel Digital Implementations of Neural Networks*. Prentice Hall, 1993.

[95] Andrew R. Putnam, Dave Bennett, Eric Dellinger, Jeff Mason, and Prasanna Sundararajan. CHiMPS: A high-level compilation flow for hybrid CPU-FPGA architectures. In *FPGA*, 2008.

[96] Rahul Razdan and Michael D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *MICRO*, 1994.

[97] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1, pages 318–362. MIT Press, 1986.

[98] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *PLDI*, 2011.

[99] Ruchira Sasanka, Sarita V. Adve, Yen-Kuang Chen, and Eric Debes. The energy efficiency of CMP vs. SMT for multimedia workloads. In *ICS*, pages 196–206, 2004.

[100] J. Schemmel, J. Fieres, and K. Meier. Wafer-scale integration of analog neural networks. In *IJCNN*, 2008.

[101] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *FSE*, 2011.

[102] Ronak Singhal. Inside Intel next generation Nehalem microarchitecture. Intel Developer Forum (IDF) presentation (August 2008), 2011. URL http://software.intel.com/file/18976.

[103] SPEC. Standard performance evaluation corporation, 2011. URL http://www.spec.org.

[104] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS*, 2004.

[105] Aater M. Suleman, Onur Mutlu, Moinuddin K Qureshi, and Yale N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *ASPLOS*, pages 253–264, 2009.

[106] S.M. Tam, B. Gupta, H.A. Castro, and M. Holler. Learning on an analog VLSI neural network chip. In *SMC*, 1990.

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

[108] Mohit Tiwari, Hassan M.G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. Complete information flow tracking from the gates up. In *ASPLOS*, 2009.

[109] Jonathan Ying Fai Tong, David Nagle, and Rob. A. Rutenbar. Reducing power by optimizing the necessary precision/range of floating-point arithmetic. *IEEE Trans. VLSI Syst.*, 8(3), 2000.

[110] N. Townsend and L. Tarassenko. Estimations of error bounds for neural-network function approximators. *IEEE Transactions on Neural Networks*, 10(2), March 1999.

[111] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ISCA*, pages 392–403, June 1995.

[112] 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. In *ASPLOS*, 2010.

[113] Ganesh Venkatesh, John Sampson, Nathan Goulding, Sravanthi Kota Venkata, Steven Swanson, and Michael Taylor. QsCores: Trading dark silicon for scalable energy efficiency with quasi-specific cores. In *MICRO*, 2011.

[114] Vicky Wong and Mark Horowitz. Soft error resilience of probabilistic inference applications. In *SELSE*, 2006.

[115] Dong Hyuk Woo and Hsien-Hsin S. Lee. Extending Amdahl's law for energy-efficient computing in the many-core era. *Computer*, 41(12), 2008.

[116] Qiang Wu, Margaret Martonosi, Douglas W. Clark, V. J. Reddi, Dan Connors, Youfeng Wu, Jin Lee, and David Brooks. A dynamic compilation framework for controlling microprocessor energy and performance. In *MICRO*, 2005.

[117] Chingwei Yeh, Yin-Shuin Kang, Shan-Jih Shieh, and Jinn-Shyan Wang. Layout techniques supporting the use of dual supply voltages for cell-based designs. In *DAC*, 1999.

[118] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. ECOSystem: Managing energy as a first class operating system resource. In *ASPLOS*, 2002.

[119] Jihan Zhu and Peter Sutton. FPGA implementations of neural networks: A survey of a decade of progress. In *FPL*, 2003.

VITA

Hadi Esmaeilzadeh will be joining the the Collage of Computing at the Georgia Institute of Technology as the first Catherine M. and James E. Allchin assistant professor of computer science. Hadi has received his Ph.D. from the Department of Computer Science and Engineering at University of Washington. He has a master's degree in Computer Science from The University of Texas at Austin and a master's degree in Electrical and Computer Engineering from University of Tehran. Hadi is interested in developing new technologies and cross-stack solutions to improve the performance and energy efficiency of computer systems for emerging applications. His research has been recognized by two Communications of ACM Research Highlights and three IEEE Micro Top Picks. Hadi's work on dark silicon has also been profiled in New York Times.