Scalable Cores in Chip Multiprocessors

by

Dan Gibson

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN - MADISON

2010

Abstract

Chip design is at an inflection point. It is now clear that *chip multiprocessors (CMPs)* will dominate product offerings for the forseeable future. Such designs integrate many processing cores onto a single chip. However, debate remains about the relative merits of explicit software threading necesary to use these designs. At the same time, the pursuit of improved performance for single threads must continue, as legacy applications and hard-to-parallelize codes will remain important.

These concerns lead computer architects to a quandary with each new design. Too much focus on per-core performance will fail to encourage software (and software developers) to migrate programs toward explicit concurrency; too little focus on cores will hurt performance of vital existing applications. Furthermore, because future chips will be constrained by power, it may not be possible to deploy both aggressive cores *and* many hardware threads in the same chip.

To address the need for chips delivering both high single-thread performance and many hardware threads, this thesis evaluates *Scalable Cores in Chip Multiprocessors*: CMPs equipped with cores that deliver high-performance (at high per-core power) when the situation merits, but can also operate at lower-power modes, to enable concurrent execution of many threads. Toward this vision, I make several contributions. First, I discuss a method for representing inter-instruction dependences, leading to more-efficient individual core designs. Next, I develop a new scalable core design, *Forwardflow*. Third, I evaluate policies by which to use Forwardflow cores in a scalable CMP. And lastly, I outline methods by which future researchers can pursue the performance characteristics of scalable cores without extensive simulation, paving the way for evaluation of system-level considerations.

Acknowledgments

Without question or hesitation, I dedicate this thesis to my beautiful and loving wife, Megan. Megan has been my support—emotional and financial—for all the years in which I have been enrolled in graduate school. In so many ways, she has implicitly and explicitly prioritized my education over her own life goals, such that I dare not dream of ever repaying her this kindness. I surely would have failed, early and often, without her unwavering support.

Naturally, thanks are due to my PhD committee: Dr. Mark D. Hill, Dr. Benjamin Liblit, Dr. Mikko Lipasti, Dr. Gurindar Sohi, and Dr. David A. Wood, my advisor. Of course, I especially wish to thank David (Wood, advisor), who oversaw my transformation from a fresh, naive, and optimistic first-year graduate student into a realistic, introspective, and objective (or perhaps more popularly, "jaded") scientist. The transformation was not easy, and quite honestly, David's management style is not suited to all students. Kevin Moore is fond of saying that "David's students must be self-motivating" (possibly not an exact quote), as David seldom micro-manages his students' daily activities. Working with David is a sink-or-swim operation: overall, a good proxy for life in general.

Like most who meet him, I found David immediately appealing for his personal charisma, and only over the course of many interactions did I come to realize the depth of recall he commands. On many occasions, I sat in (silent) awe of his ability to traverse layers of abstraction. I wonder if this gradual exposure is intentional on David's part—a method of appearing more approachable to mortals? It seems plausible. Most of all, I respect David as, certainly, the deepest fount of intellect that I have ever had the privilege to know. I'm honored to have been his student.

It is impossible to mention David without also mentioning Mark Hill (my "academic uncle", to adopt his parlance). In 2005, I was a lowly graduate student in Mark's CS 757 class. My interests at that time were in VLSI, and I was happily and gainfully involved in a research group in ECE, studying that area. Unfortu-

nately, that arrangement began to disintegrate, for political reasons beyond my control (and beyond my understanding at the time). It would be inaccurate to say that I was desperate to find another advisor, but I was certainly worried, and at that time strongly considered leaving graduate study. It was then that Mark recruited me to work for David, albeit on a temporary basis, who was away on sabbatical at the time and unable to recruit students of his own. In essence, I owe my career in architecture to Mark's successful recruitment effort. During my time with the Multifacet group, Mark's advice on presentations, writing, and human interaction in general was useful beyond measure.

Aside from my committee, several others belonging to the Wisconsin faculty deserve thanks, for their outstanding efforts at teaching, and enlightening conversations over the years. There are too many to list exhaustively, but certainly I wish to thank explicitly Mike Swift and Mary Vernon, from whom I learned a great deal. I also was the recipient of much technical help and assistance from the lab, CSL, to whom I am very grateful. In particular, I wish to thank John Perkins for explaining many details of practical Solaris administration to a wide-eyed newbie, and Tim Czerwonka for volunteering (more than once) to service cluster nodes on weekends and holidays, when deadlines loomed.

Many past student colleagues served as positive role models, helping to tune my ever-changing expectations. Wisconsin PhDs Bradford Beckmann, Alaa Alameldeen, Kevin Moore, Min Xu, Natalie D. Enright Jerger, Mike Marty, Phillip Wells, Luke Yen, Jayaram Bobba, and Matthew Allen, in their various capacities, encouraged me to stay the course and follow their superb examples. I am humbled to join that elite crowd. From them, I learned not to allow simulators, papers, and deadlines to dominate my life (at least not at all times), and how to handle the frequent rejections, ever-present in this field, with grace. Thank you, everyone.

I've also had the pleasure of working with concurrently with others who I don't consider to be "more senior" than myself; it would be more accurate to say that I consider them contemporaries. These certainly include Derek Hower, Spyros Blanas, Haris Volos, and Marc DeKruijf. Each of these students represent the

best of what science and engineering can make, in their various areas of expertise. I thank them for challenging my views, as they are quick to harden and difficult to change. I'm also very glad that my immediate employment does not take me away from Madison, as I can continue to pursue my relationships with these most excellent friends.

Among all my peers, I've had the chance to interact most closely with Yasuko Watanabe, collaborating together on common infrastructure and a common research project. I will miss those collaborations, as it is a very, very rare thing to find so able a collaborator as she. Research in our particular sub-area is a mind-numbing, unrewarding, and difficult process, and we have suffered through its pitfalls together. I'm glad to have had the company, and I happily entrust the future of our somewhat-conjoined work to her keeping.

I see a great deal of myself in my junior colleagues, Somayeh Sardashti, Arkaprava Basu, Rathijit Sen, and Matt Sinclair. I've tried to help them as others have helped me, to continue the chain of graduate student wisdom as best I can. Of course, I wish them the best and will watch their progress—hopefully my example will be as useful to them as my senior graduate student mentors were to me.

In the next chapter of my life, I have the privilege of joining an excellent group of people at Google's Madison office. In the course of two internships, I came to greatly respect my past-and-future colleagues, for their technical prowess and awesome degrees of confidence. I especially wish to thank Dennis Abts, Peter Klausler, Jim Laudon, Florentina Popovici, Kyle Nesbit, Andy Phelps, Mike Marty (again), Phillip Wells (again), and Polina Dudnik, for an experience that convinced me to spend at least the next few years working with them again.

Without a doubt, the comforts of friends, in and out of this field, made the PhD experience more tolerable. Jake Adriaens and I commiserated on many occasions over our many trials, and despite our lives having taken different paths for the moment, I am eager to see them reconverge in the coming year. I can think of no one living person with whom I would rather work, on any project, at any time. I wish him the best of luck finishing his own thesis, as I'm sure he has his hands rather full, with his two newborn daughters. I also

wish to specifically thank Tim Dagel, my oldest friend, for the many years of amusement past and to come. Tim knows there were times at which the only force holding me together was his friendship, and I am forever indebted to him for it.

I wish to acknowledge my family (blood relatives and otherwise). Chief among them is my mother, who I hope will read this acknowledgment with pride. Mom: You may want to skip the rest of the thesis—*this* is the good part. To my dear sister, Jamey, and my brother, Dean, I thank you for your encouragement, support, advice, and affections. I ask only that you not frequently call me "Dr. Gibson", as I reserve that title forever in my memory for our father.

Lastly, I acknowledge my father, who always encouraged me to *be a lifelong student* (so far, this has literally been the case). When he passed away in 2009, I discovered just how much of my own motivation was derived from a personal need to inspire his feelings of pride in my accomplishments. For a time after his death, I doubted whether I had any desire of my own to pursue a PhD. Somehow, my wife, family, and friends managed to convince me to persevere.

In some of my darkest and most unsure moments, I would read from two of his research papers[1] [2], hoping to draw inspiration from that connection. Though our chosen fields of study are very different, an occasional reminder of a family legacy grounded in science was comforting in the face of many trials. Thank you, sir, for encouraging me to pursue a challenging and interesting field, for supporting me when I needed it, for setting a lofty example to follow, and for being a friend *and* a father.

1. Thompson, Carter, Gibson, Reiswig, and Hinshaw. Acute Idiopathic Retroperitoneal Fibrosis. From the Department of Surgery, College of Medical Evangelists School of Medicine and the Los Angeles County General Hospital. 1960.
2. Gaspar and Gibson, Herniotomy for Umbilical Hernia in the Bad-Risk Patient: Report of 16 Cases. From the Department of Surgery, College of Medical Evangelists and the Los Angeles County General Hospital. 1959.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer architecture is now facing the “Fundamental Turn Towards Concurrency” [162]. After years of dramatic improvements in single-thread performance, chip manufacturers’ race to increase processor clock frequency has hit a wall—the so-called *Power Wall*. Now, *Chip Multiprocessors (CMPs)* are emerging as the dominant product for most major manufacturers, signalling the start of the *multicore era*. Architects hope to reap future performance improvements through explicit software-level concurrency (i.e., multi-threading). But if future chips do not address the needs of today’s single-threaded software, designers may never realize the vision of highly-threaded CMPs.

Architects cannot afford to ignore single-thread performance. Some effort must be devoted to improving individual cores, despite power and thermal limitations. However, if the eventual vision of highly-parallel software is ever to come to fruition, future CMPs must also deliver more hardware concurrency, to encourage software designers to specify parallelism explicitly. Together, these demands motivate *Scalable Cores*, cores than can easily adapt to single- and multi-threaded workloads, according to the current system-wide demand. This work considers the design of scalable cores, and the tradeoffs involved in their deployment in future CMPs.

I begin this work with a brief discussion of the chip manufacturing landscape leading to the widespread adoption of CMPs, and the potential problems to be expected in future chip generations. I demonstrate how the needs of future CMPs can be addressed with *scalable cores*—cores

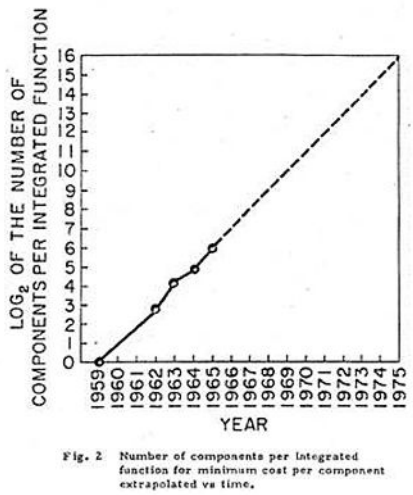


FIGURE 1-1. Gordon Moore's plot of the number of components (log scale) in a fixed are per year, courtesy of Intel®'s press kit.

that can trade off power and performance as the situation merits to best suit a particular workload. The design of such cores and the policies surrounding their design and use are the principal foci of my work in this thesis. I briefly outline the main contributions of this work, and conclude this chapter with a roadmap of the remainder of the document.

1.1 The Emergence of Chip Multiprocessors

The last several years have witnessed a paradigm shift in the microprocessor industry, from chips holding one increasingly complex out-of-order core to chips holding a handful of simpler cores [84, 167]—*Chip Multiprocessors* (CMPs). Despite the promise of more transistors [51], power and thermal concerns have driven the industry to focus on more power-efficient multicore designs. Microarchitects hope to improve applications' overall efficiency by focussing on thread-level parallelism (TLP), rather than instruction-level parallelism (ILP) within a single thread. Some researchers project that this trend will continue until chips have a thousand cores [15].

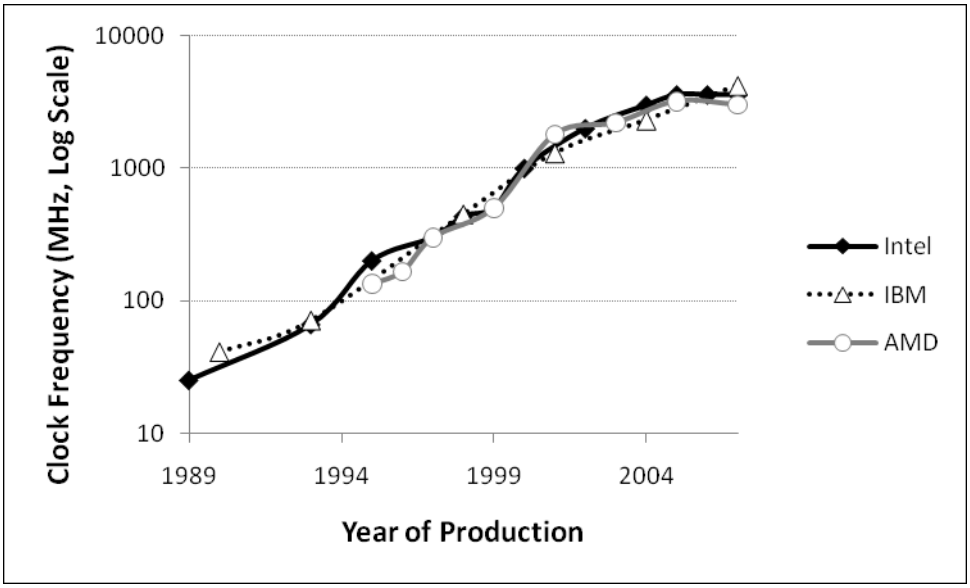


FIGURE 1-2. Clock frequency (logarithmic scale) versus year of product release from 1989 through 2007.

1.1.1 Moore's Law

Gordon Moore introduced in his 1965 article in *Electronics* [116] a trend that would shape the computing industry for decades to come. Commonly called *Moore's Law*, Moore observed that per-transistor manufacturing cost would fall predictably for at least the coming years, resulting in a great increase in the number of transistors feasible to include in a single integrated circuit (illustrated by Moore's own plot in Figure 1-1). Moore's predictions held for *decades*, and rapid lithographic improvements yielded an exponential increase in the number of transistors on a die. Chip manufacturers leveraged smaller and more numerous devices to deliver a commensurate improvement in single-thread performance (sometimes called "Popular Moore's Law"), driven in part by increasing processor clock frequency. Figure 1-2 plots the growth of clock frequency across major

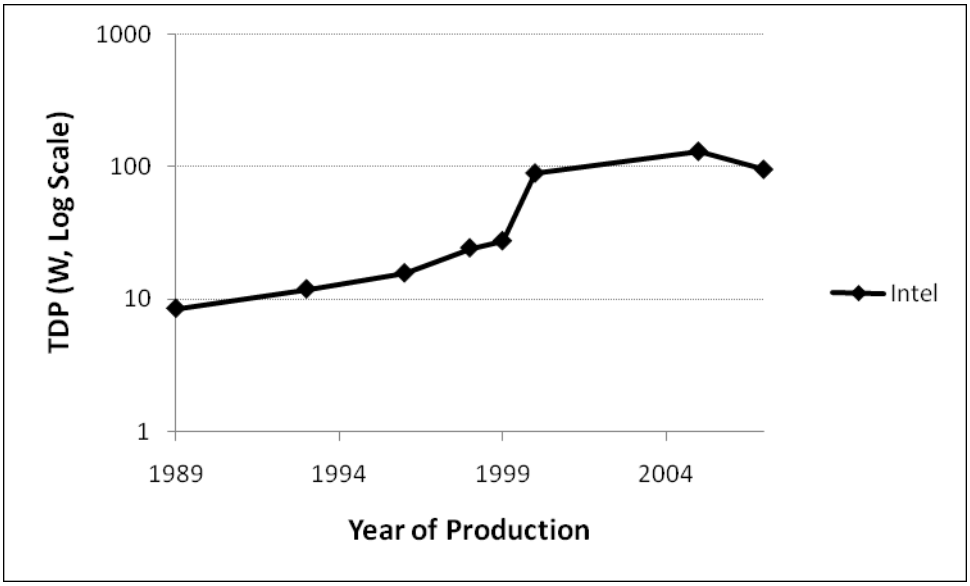


FIGURE 1-3. Thermal design power (TDP) (logarithmic scale) versus year of product release from 1989 through 2007 (Intel® products only).

chip manufacturers' mainstream product offerings [138, 11, 83, 48]. Until recently, clock frequency grew dramatically and quickly for all major manufacturers.

This rapid growth in frequency all but halted when chips hit what has been dubbed the *Power Wall.* As frequencies rose, so too did dynamic operating power, driven by the dual effect of more devices (Moore's Law) and faster clock frequency (Popular Moore's Law). Rising dynamic power consumption meant that chips started to reach the practical limits of cost-effective cooling technologies. The heat produced by a chip is it's *thermal design point (TDP),* plotted in Figure 1-3 for Intel® mainstream offerings. As a result of the Power Wall, growth in clock rate stagnated, TDP has remained in the vicinity of 100 watts, and frequency has held near 3 GHz for the last several years.

Unable to leverage further increases in frequency to drive performance improvements for single threads, architects instead sought to deliver additional computation throughput through the

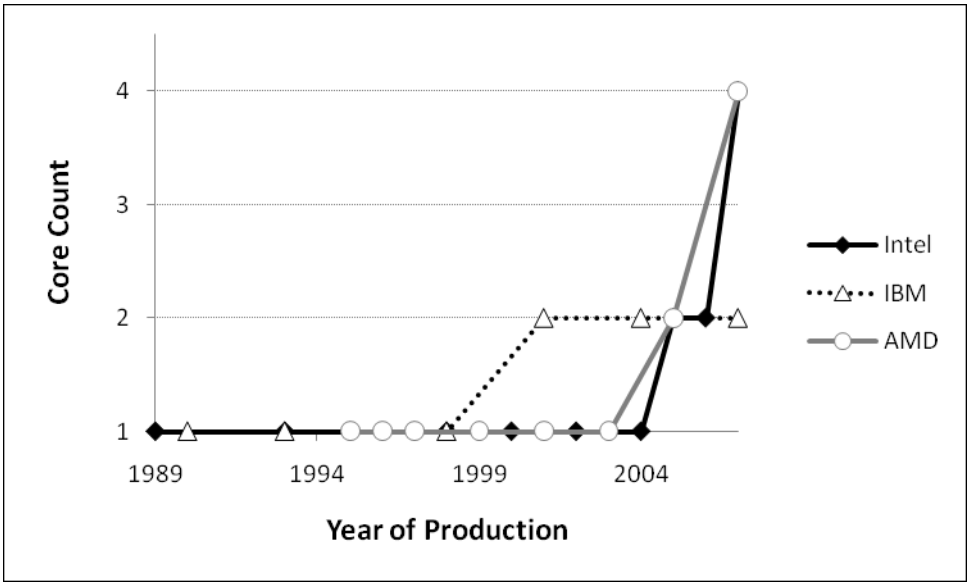


FIGURE 1-4. Core count versus year of product release from 1989 through 2007.

incorporation of multiple processing cores on a single die. Figure 1-4 plots this trend for the same products used in Figure 1-2. By the end of 2007, *Chip Multiprocessors* (CMPs) were mainstream product offerings of all major chip manufacturers.

1.2 The Challenge of CMPs

At least two fundamental problems undermine widespread success of CMPs. First, the Power Wall remains, and though the era of aggressive increases in frequency appears to be over, Moore's Law endures and future chips will be severely power-constrained. Second, CMPs rely on explicit software parallelism, which is not (yet) pervasive in the software industry.

1.2.1 The *Utilization Wall*

The physical limits on power delivery and heat dissipation that led to the Power Wall still exist. In the long term, to maintain fixed power and area budgets as technology scales, the fraction

of *active* transistors must decrease with each technology generation [32, 1, 173], a trend Ahn et al. coin the *Utilization Wall*.

The most important implication of the Utilization Wall is that the Simultaneously Active Fraction (SAF)—"the fraction of the entire chip resources that can be active simultaneously" [31]—is already less than one[1] and will diminish further in future technology generations. Designers of future CMPs must consider falling SAF as a first-order design constraint, in addition to other low-power design considerations. In particular, future chips must expect to opportunistically power-off unused resources (e.g., cores, caches, hardware accelerators, etc.).

Ultimately, future chips may simply consume more power than those of today—it is electrically and mechanically possible to build such chips with current technologies, albeit at greater costs than current systems incur for power distribution and heat removal. However, for every pin on the packaging used to deliver current to the chip or remove current from the chip, available off-package bandwidth for communication is potentially reduced. Improved technology will grow memory bandwidth (e.g., with high-speed differential signalling [76] or FB-DIMM [57]), but intuitively, as CMPs deliver more and more execution contexts, they will require more off-package bandwidth to service memory and I/O demand.

1.2.2 Finding Parallelism

The second trend undermining the vision for CMPs is the lack of abundant parallel software. In the short term, both servers and client machines have been successful markets for CMPs, because contemporary CMPs integrate only a handful of cores per die. To some extent, both cli-

1. For example, in the *Nehalem* CPU, not all cores can operate simultaneously at full speed [170].

ent- and server-class machines can *scale-out*. That is, they can leverage additional cores in CMPs to perform independent work with greater computational throughput, without the expectation or requirement of actually decreasing request latency or improving perceived single-thread performance. This is related to *weak scaling*, from the high-performance computing community, in which additional processors can be used to solve larger instances of the same problem in the same overall runtime.

Scale-out contrasts with *strong scaling*, in which runtime shrinks with the addition of additional computation resources. The performance improvements during the era of rapid frequency growth (enjoyed by clients and servers alike) behaved more like strong scaling than weak. As such, the degree of success to be expected from the scale-out approach is yet to be seen. While some applications scale out with enormous success [21, 59, 42], important classes of workloads remain that scale-out poorly or not at all (e.g., legacy single-threaded applications).

Ultimately, the effectiveness of server scale-out will endure so long as computational needs are insufficient to saturate other resources (e.g., main memory bandwidth)—and provided independent work can be efficiently routed to available thread contexts. Client scale-out is a more pressing concern, as client-side processing is thought to exhibit less task-level parallelism than do servers.

Further complicating the problem, CMP vendors can expect little help from entirely new software. Despite decades of research on parallel programming and abstractions, including renewed interest in recent years [e.g., 117, 102, 66, 110, 13, 188, 36, 131, 189, 115, 77, 10], commonplace parallel programming remains unrealized, and continues to be a largely unsolved problem for a variety of difficult-to-parallelize applications.

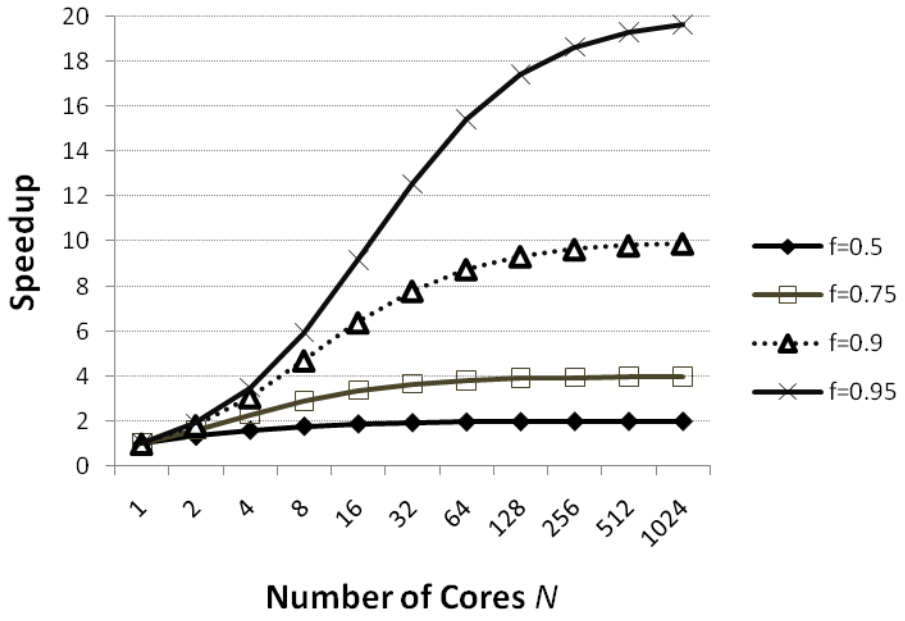


FIGURE 1-5. Speedup as a function of the number of cores *N*, as predicted by Amdahl's Law.

1.2.3 Amdahl's Law

Regardless of the approach used to find parallelism, Amdahl's Law [12] still applies: The speedup of a parallel execution is determined by the fraction of the total work that can be executed in parallel, f, and the degree of parallelism, N (Equation 1.1).

$$Speedup = \frac{t_{new}}{t_{old}} = \frac{1}{(1-f) + \frac{f}{N}} \tag{1.1}$$

Even well-parallelized applications have sequential bottlenecks that limit their parallel speedup (and many applications are not currently parallel at all, i.e., $f = 0$). Figure 1-5 plots expected speedup as a function of core count (i.e., N), over four values of f. As evidenced by the

plot of $f = 0.95$, sequential bottlenecks eventually dominate overall performance, even with a huge degree of available hardware parallelism (e.g., a paltry speedup of 20x is observed even on a thousand cores). In other words, a thousand simple cores may maximize performance in an application's parallel section, but simple cores exacerbate sequential bottlenecks by providing limited instruction-level parallelism (ILP). Hill and Marty [73] argue that Amdahl's Law leads to the conclusion that "researchers should seek methods of increasing core performance even at high cost." In other words, architects must not focus only on increasing thread-level parallelism (TLP), but should seek a balance between TLP and ILP.

1.2.4 Of Walls, Laws, and Threads

Amidst all these trends, interesting times lie ahead for architects. Technology improvements will continue to yield more transistors (Moore's Law), but power and thermal constraints will prevent architects from using all of those devices at the same time (Utilization Wall). CMPs offer the promise of increased computation throughput, but single-threaded applications and serial bottlenecks threaten to limit parallel speedup (Amdahl's Law). Architects cannot entirely avoid such bottlenecks, because imprudent exploitation of ILP may cause TDP to rise beyond the heat-dissipation capability of cost-effective cooling techniques (Power Wall).

1.3 Core Design for ILP and TLP: *Scalable Cores*

Together, ample transistors, flat power budgets, and demand for both threading and single-thread performance motivate *scalable cores*: cores that can trade off power and performance as the situation merits. Concretely, scalable cores have multiple operating configurations at varied power/performance points: they can *scale up*, allowing single-threaded applications to aggres-

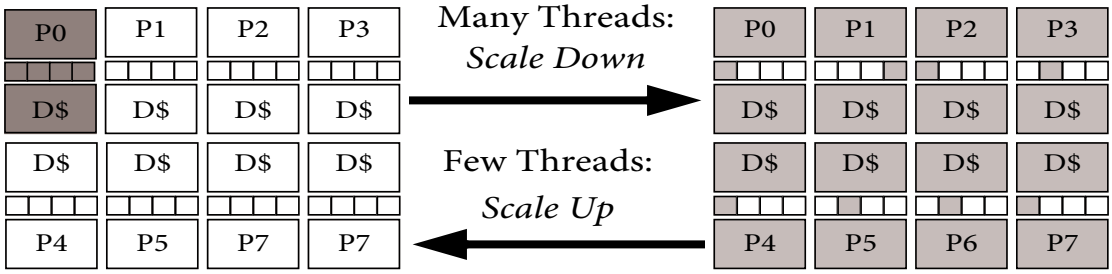


FIGURE 1-6. CMP equipped with scalable cores: Scaled up to run few threads quickly (left), and scaled down to run many threads in parallel (right).

sively exploit ILP and memory-level parallelism (MLP) to the limits of available power, or can *scale down* to exploit TLP with more modest (and less power-hungry) single-thread performance.

Figure 1-6 illustrates a conceptual CMP equipped with scalable cores under two different operating scenarios. On the left, the CMP is shown with two cores fully scaled up, indicated by greyed (powered-on) processors and components. Under this configuration, the CMP runs only one thread at a time[2], but at maximum single-thread performance. On the right, the CMP has activated the other processing cores, and may now run many more threads. But, in order to keep the chip-wide power budget within acceptable bounds, each processor is individually scaled down by powering-off a subset of its available resources.

Heterogeneous CMPs approach this vision by scaling cores *statically*, provisioning some cores with more resources and some with less [99]. Composable core designs scale power and performance by *dynamically* merging two or more cores into a larger core [94, 86], but incur substantial latency penalties when interacting with distant resources (i.e., those that reside in the

2. Effectively, a scaled-up CMP becomes an overprovisioned multi-core system [31].

opposite core). By appropriately mapping threads and enabling cores, both approaches represent initial steps towards a truly scalable core design.

On the other hand, dynamic voltage and frequency scaling (DVFS) techniques deliver behavior similar to what might be expected from a scalable core, as typified by designs like Intel's ® *Nehalem* [85]. However, dwindling operating voltage suggests the effectiveness of DVFS will be limited in future chip generations [191, 180]. In the post-DVFS regime, *scaling core performance means scaling core resources* to extract additional ILP and MLP—either by statically provisioning cores differently or by dynamically (de)allocating core resources.

Therefore, in a scalable core, *resource allocation changes over time*. Cores must not rely on powered-off components to function correctly when scaled down, and must not wastefully broadcast across large structures when scaled up. Conventional microarchitectures, evolved in the domain of core-private and always-available resources, present significant challenges with respect to dynamic core scaling. Designers of scalable cores should avoid structures that are difficult to scale, like centralized register files and bypassing networks. Instead, they should focus on structures that can be easily disaggregated, and powered-on or off incrementally to adjust core performance independent of other structures.

1.4 Contributions

This thesis makes contributions in two major areas: a new core scalable core design (*Forwardflow,* Section 1.4.1), and an investigation of policies suitable for deployment of scalable cores in future CMPs (Section 1.4.2).

1.4.1 Forwardflow: A Statically- and Dynamically-Scalable Core

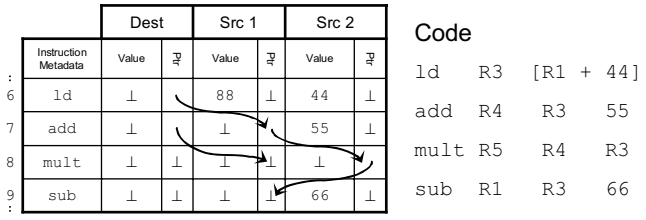


FIGURE 1-7. Dataflow Queue Example

Conventional core microarchitectures have evolved largely in the uniprocessor domain, and scaling their microarchitectural structures in the CMP domain poses significant complexity and power challenges. The first main contribution to this work presents a method for representing inter-instruction data dependences, called *Serialized Successor Representation* (SSR, Chapter 4). SSR represents inter-instruction dependences via a linked list of *forward pointers* [132, 136, 176]. Chapter 5 follows with a discussion of a core design leveraging SSR, the *Forwardflow Architecture*. Forwardflow is a scalable core design targeted at power-constrained CMPs leveraging a modular instruction window. Instructions, values, and data dependences reside in a distributed *Dataflow Queue* (DQ), as illustrated in Figure 1-7. The DQ is comprised of independent banks and pipelines, which can be activated or de-activated by system software to scale a core's execution resources to implement core scaling.

Forwardflow cores are highly energy-efficient—they are the most efficient design overall in greater than 90% of studied benchmarks. With respect to a traditional microarchitecture, a Forwardflow core offers comparable performance at a given window size, but at reduced per-core power consumption. For instance, in a 128-entry instruction window design, Forwardflow cores consume about 12% less *chip-wide* power, compared to a traditional out-of-order microarchitecture (a substantial decrease in *per-core* power), while maintaining about the same performance. Larger Forwardflow cores (i.e., 256-entry windows) reduce runtime by 14% on average, requiring

no more power than that consumed by a traditional microarchitecture. Forwardflow cores scale further still, delivering about 10% runtime reduction for each doubling of window size.

1.4.2 Scalable Cores in CMPs

This thesis also investigates policies for designing and using scalable cores, in which the Forwardflow core microarchitecture serves as a vehicle for evaluation. In Chapter 6, I discuss tradeoffs between *borrowing* designs, in which cores scale up by borrowing scalable resources from nearby cores, and *overprovisioned* designs, in which all scalable resources are core-private and provisioned for the largest possible configuration. I show that scalable Forwardflow cores are cheap to overprovision (i.e., 7% increase in die area), and that borrowing can lead to performance problems if borrowed resources are only slightly slower than overprovisioned resources. The wisdom of "Loose Loops Sink(ing) Chips" [23] holds for scalable cores.

Next, I propose and evaluate several hardware scaling policies, for operation below the level of the operating system scheduling slice. I demonstrate a simple mechanism which can be used by system software to approximate the most energy-efficient core configuration when running single threads. When running many threads, I show that previously-proposed spin-detection hardware [182] can be used as an indicator to dynamically *scale cores down*, thereby increasing the overall efficiency of the execution. I also show that programmer-guided scaling is a simple means by which sequential bottlenecks can be identified. Scalable cores can use programmer-supplied annotations to trigger scaling decisions.

Lastly, I lay the groundwork for future software-driven policy work, by showing that DVFS adequately models the performance of a scalable core, provided a sufficiently wide group of benchmarks are studied. Though DVFS does not model the same power consumption, I show that

per-configuration energy consumption can be estimated accurately using correlated event counts, e.g., hardware event counters. This final contribution indicates that future researchers need not be burdened by the development time, long run times, and frustrations of simulation-based evaluation, paving the way for future work in software policies for scalable CMPs.

1.5 Thesis Organization

Chapter 2 discusses background material, trends, further motivation, and prior work in the area of scalable core design. Chapter 3 details the quantitative evaluation methods used in this dissertation, including details of simulation infrastructure, workloads, and other modeling methods. Chapter 4 introduces SSR, and Chapter 5 discusses the Forwardflow core implementation. I discuss and evaluate policies for implementing core scaling decisions in Chapter 6, and conclude this dissertation in Chapter 7 with a re-summarization of content, key results, possible future work, and reflections.

The material of Chapters 4 and 5 differs from my other published work in detail and organization. Chapter 4 considers the implications of pointer-based representation in isolation, and Chapter 5 gives a great deal of detail on the Forwardflow design itself, with less emphasis on placing that design in a wider CMP setting. Chapter 5 also gives many more details of the Forwardflow implementation used in this dissertation, which is similar to but not identical to that of the ISCA 2010 paper [60].

Chapter 2

Scalable Cores, Background, and Related Work

This chapter outlines vision for scalable cores and gives an overview of related work. I begin by introducing microarchitecturally-scaled cores. Next, I explain why dynamic voltage and frequency scaling (DVFS) will not suffice to provide scalable core functionality—instead, *microarchitecture* must scale. I discuss high-level tradeoffs inherent in the design of scalable cores. I then present relevant background on contemporary microarchitectures, leading to a general discussion of related work.

2.1 A Vision for *Scalable Cores*

A *scalable core* is a processor capable of operating in several different configurations, each offering a varied power/performance point. A configuration may constitute an operating frequency, an active set of microarchitectural resources, or some combination of both. Scalable cores are compelling because, when the power to do so is available, scalable cores can *scale up*, allowing single-threaded applications to aggressively exploit ILP and MLP. Or, when power is constrained, scalable cores can *scale down* to conserve per-core energy, e.g., to exploit TLP with more modest (and less power-hungry) single-thread performance. In other words, scalable cores have the potential to adapt their behavior to best match their current workload and operating conditions. Figures 2-1 and 2-2 illustrate two operating extremes (single- and highly-threaded), along with the approach to each operating scenario used by today's CMPs.

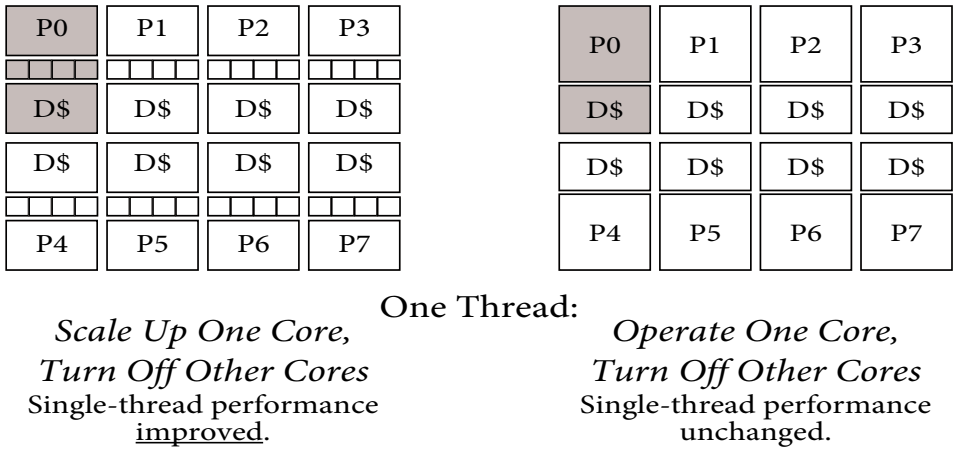


FIGURE 2-1. Scalable CMP (left), one core fully scaled-up to maximize single-thread performance. Traditional CMP, operating one core.

In the first case, a single-threaded application, the scalable CMP in Figure 2-1 (left) adapts to its current workload by *scaling up* one core (indicated by gray shading), and halting other unused cores to conserve energy (indicated by white backgrounds). The traditional CMP (right) halts unused cores, but with no scalable components, it has no microarchitectural ability to address single-threaded performance.

In the second case, a multi-threaded application (or collection of single-threaded applications), the scalable CMP in Figure 2-2 *scales down* all cores. Scaling down causes individual cores to consume less power, and also reduces their demands on the memory subsystem. As a result, overall chip-wide power consumption is reduced, as compared to scaled-up configurations. The traditional CMP (right) has no scale-down mechanism and therefore runs the risk of drawing too much power when operating at full load. Consequences of sustained operation above the

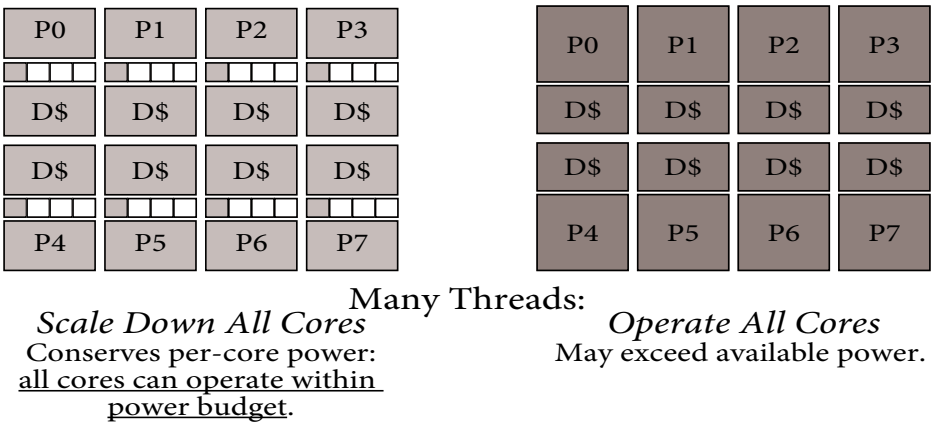


FIGURE 2-2. Scalable CMP (left), all cores fully scaled-down to conserve energy while running many threads. Traditional CMP, operating all cores.

power budget include circuit instability (e.g., from insufficient current supply affecting rail voltage) and overheating concerns.

2.1.1 Dynamic Voltage and Frequency Scaling (DVFS)

Intel's *Nehalem* already exhibits some of the desired behaviors of a scalable core, using its frequency scaling "turbo mode" to accelerate single threads [34, 170]. By using a faster clock for turbo mode, computational throughput from a single thread is increased. *Nehalem*'s turbo mode is an example of the more general *dynamic frequency and voltage scaling (DVFS)* technique.

DVFS techniques for increasing frequency rely on the fact that, in CMOS, the rate at which logic circuits operate is determined by the rate at which capacitances (i.e., transistor gates and parasitic capacitances) charge (to logic "1") or discharge (to logic "0"). One can approximate the time required to charge or discharge these capacitances as proportional to the inverse of the operating voltage [145]. Therefore, by increasing operating voltage, a circuit's switching speed can be

increased proportionately. In other words, *increase in voltage enables a linear increase in operating frequency.*

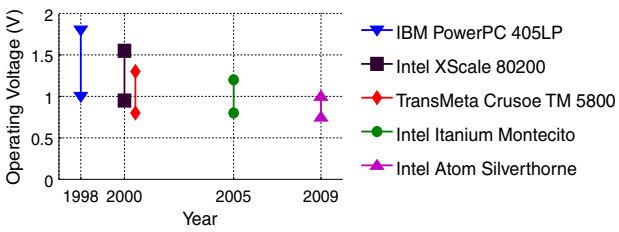


FIGURE 2-3. Operating voltage range over time.

However, architects cannot simply rely on DVFS to implement the scale-up functionality of scalable cores in future product generations. Figure 2-3 plots the recent trend of operating voltages. As technology scales, the dynamic operating voltage range in each technology generation shrinks [52, 180]. This trend arises because, while minimum supply voltage, V_{DD-min}, has decreased only slightly, maximum supply voltage, V_{DD-max} has dropped steadily.

One overarching concern leads to limits on dynamic voltage range: power consumption. The drive to reduce V_{DD-max} is derived from its quadratic relation to dynamic switching power. *Reductions* in V_{DD-max} yield square reductions in switching power. On the other hand, *increasing* supply voltage scales up dynamic power, again quadratically. When this voltage increase is paired with a frequency increase (e.g., under DVFS turbo mode), the increase in dynamic power is cubic.

Leakage concerns constrain V_{DD-min}: leakage current increases *an order of magnitude* for each 100mV decrease in device threshold voltage. In order for devices to operate at all, designers must maintain $V_{DD-min} > V_t$ (preferably by a healthy margin, to maintain clean signalling levels [191]). In other words: device manufacturers cannot afford to further lower threshold voltage, to make room for lower V_{DD-min}. Effectively, circuit designers are stuck between a rock (inability to lower V_{DD-min}) and a hard place (the necessity to lower V_{DD-max}). Together, these trends suggest that

total operating voltage range will be fairly limited as technology scales.[1] With reasonable operating voltage ranges diminishing, the amount of upward scaling possible from DVFS will shrink as well.

It is important to note, however, that operating frequency can be safely *scaled down* without changing operating voltage (i.e., DFS). Future designs will likely continue to rely on frequency scaling to conserve dynamic energy. However, one must consider that DFS does not affect static power—clocks can be scaled down arbitrarily, but leakage remains unchanged.

2.1.2 Scalable Microarchitecture

Without DVFS, *scaling core performance means scaling core resources* to extract more instruction-level and memory-level parallelism from a given workload (ILP and MLP, respectively). Therefore, in a scalable core, resource allocation changes over time—determining exactly which resources scale depends on the details of a particular implementation. In general, cores must not rely on powered-off components to function correctly when scaled down, and must not wastefully broadcast signals across large structures when scaled up. Designers of scalable cores should avoid structures that are difficult to scale, like centralized register files and bypassing networks. Instead, they should focus on structures that can be easily disaggregated, and powered-on incrementally to improve core performance independent of other structures.

To compete with traditional designs, a scalable core should have a nominal operating point at which it delivers performance comparable to a traditional out-of-order core at comparable power, and should offer more aggressive configurations when scaled up. That is, performance itself should not be sacrificed for performance scaling. A wide performance/power range is desirable,

1. Deepaksubramanyan and Nunez offer a useful analysis of leakage current's relationship to technology parameters and operating conditions [43].

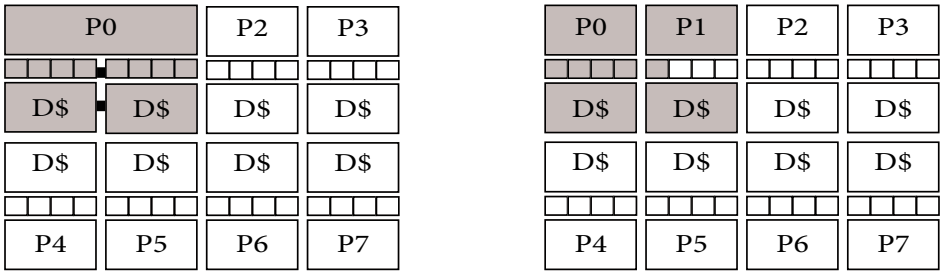


FIGURE 2-4. Resource-borrowing scalable CMP (left) and resource-overprovisioned CMP (right).

but scaling up should be emphasized over scaling down (as the latter can be easily accomplished with DFS).

Canonical work towards scalable CMPs, *Core Fusion* [86] and *Composable Lightweight Processors* [94], compose entire cores to scale all pipeline resources at the same rate. These designs adopt an implicit philosophy of *resource borrowing*—composed or fused cores aggregate resources from other cores to implement scale-up. This approach is sensible, so long as area cost of scalable components is high (i.e., it is worthwhile to amortize area cost via sharing). However, borrowed resources are more likely to lengthen wire delay, as borrowed resources reside within the boundaries of other cores. As an alternative to borrowing, scalable cores can be individually *overprovisioned*, assigning additional per-core (private) resources to each scalable core, which are never shared with other cores during scale-up. When scaled down, cores simply leave these extra resources unutilized (or unpowered, if static consumption is a concern). This *resource overprovisioning* philosophy makes sense if scalable core components consume small on-die areas (i.e., cost of overprovisioning is small, compared to the latency cost of communication with other cores). Figure 2-4 illustrates the differences between the borrowing (left) and overprovisioning (right) philosophies: Borrowing potentially brings more chip area to bear when operating in a sin-

gle- or lightly-threaded mode, but requires sharing of resources to do so. Overprovisioning potentially wastes area when scaled down (e.g., *P1* in the figure), but each core can scale independently of others.

2.1.3 Realizing the Vision

The foci of this work is the realization of the above vision: how to build an efficient, high-performance scalable core, and how to effectively use such a core in a scalable CMP. Before I discuss the technical content of this thesis, I first provide background information detailing the current microarchitectures in CMPs. I also take the opportunity to discuss prior work by others.

2.2 Background

Little's Law suggests that a core must maintain enough instructions in flight to match the product of fetch width and the average time between dispatch and commit (or squash). These buffered instructions constitute an *instruction window*—the predicted future execution path. As memory latencies increase, cores require large windows to fully exploit even modest fetch bandwidth. Much prior work has built on this observation by focusing on scaling window size, e.g., to expose parallelism in memory-intensive workloads.

However, not all instructions in the window are alike. In a typical out-of-order design, instructions not yet eligible for execution reside in an *instruction scheduler*. The scheduler determines when an instruction is ready to execute (wakeup) and when to actually execute it (selection), based on operand availability. In general, instruction *windows* are easily scalable because they are SRAM-based, while many instruction *schedulers* are not because they rely on CAM-based [190] or matrix-based [70, 144] broadcast for wakeup and priority encoders for selection.

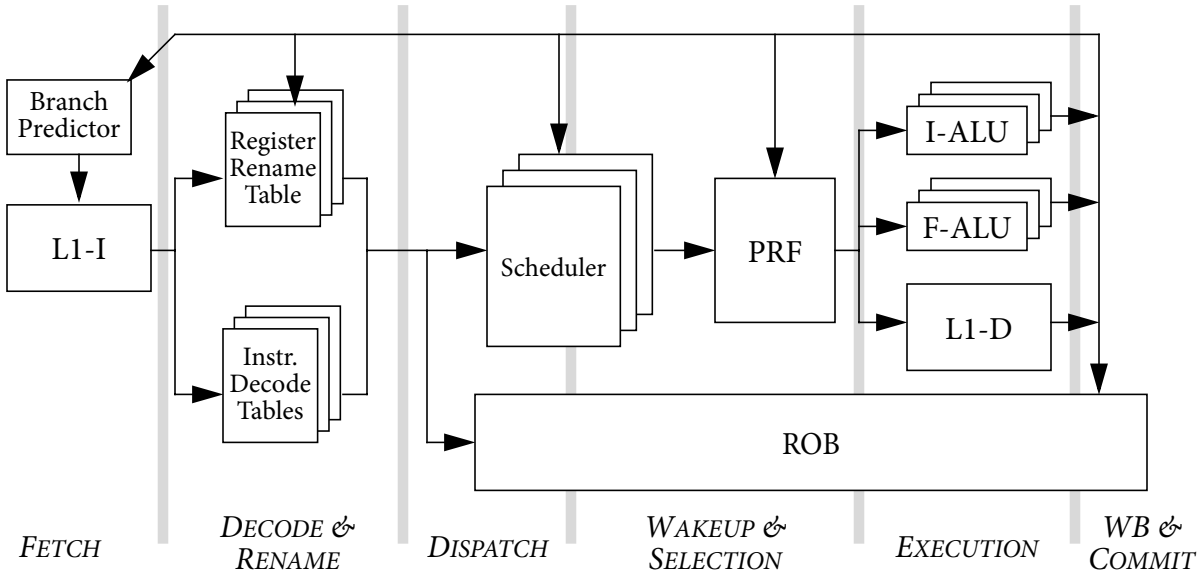


FIGURE 2-5. "Canonical" out-of-order microarchitecture.

2.2.1 A Canonical Out-of-Order Superscalar Core

Figure 2-5 is a basic block diagram of (what I call) a "canonical out-of-order superscalar core." In relation to the microarchitectures of industrial designs, it is most similar to the DEC/Alpha product offerings, e.g., the Alpha 21264 and 21364 (EV6 and EV8, respectively) [88, 41], but the general pipeline format is widely applicable. Though the diagram suggests a seven-*cycle* pipeline, each enumerated pipeline stage is typically itself pipelined. In other words, this discussion considers a seven-*stage* pipeline, and attempts to be agnostic of the precise number of cycles required by each.

To begin, *Fetch* anticipates the future set of instructions (i.e., branch prediction) and retrieves these instructions from instruction memory, in predicted program order. In a superscalar core, *Fetch* aims to deliver W instructions per cycle to the decode pipeline, where W is the overall fron-

tend[2] pipeline width (it is a common practice for all pipeline stages' widths to match, preventing any one stage from becoming a bottleneck). Because cache line widths tend to be larger than *W* (when measured in instructions), *Fetch* is implemented via a bubble-fetch process, in which the L1-I cache is read only frequently enough to deliver new instructions. During contiguous fetch operations, the previously-read line is buffered, and sources the next instruction in predicted program order. During non-contiguous fetch operations (i.e., in the presence of branching, very common), instructions from a single cache line may appear out of order with respect to memory address (e.g., in the case of a followed back-branch) in a given cycle's *W* instructions, or even redundantly (e.g., within a tight loop). In short, the fetch logic is responsible for properly sequencing instructions, according to the rules set forth for doing so by the ISA[3].

Occasionally, conditions arise that prevent *Fetch* from flowing all *W* instructions to *Decode*. For instance, an instruction fetch operation may incur a miss, during which *Fetch* makes no progress. Similarly, a miss in the I-TLB, filled in software, causes a stall. Additionally, port counts on the L1-I cache may prevent *Fetch* from following arbitrary code sequences in a single cycle—even if they are perfectly predicted.

Once fetched, instructions propagate to the *Decode & Rename* sections of the pipeline. These operations identify inter-instruction data dependencies, translate instruction opcodes to appropriate control signals and metadata, and resolve write-after-write dependences via *Register Renaming*[4]. Though Figure 2-5 suggests instruction decode and register renaming are independent, it is

2. I.e., *Fetch*, *Decode & Rename*, and *Dispatch*.

3. For instance, in SPARCv9, *Fetch* must handle branch delay slots, by fetching the instruction following some branches, and flowing the delay-slot instruction to the decode stage immediately behind its corresponding branch.

4. Sima gives an overview of renaming techniques in a 2000 *IEEE Computer* article [150].

valuable to precede rename with non-speculative register identification, to reduce port and speculation requirement on the renamer itself.

The Register Rename Table ("renamer," sometimes called the Register Alias Table [e.g., 26]) is a critically-constrained resource in an out-of-order core. The renamer translates architectural register names into physical register identifiers, allowing independent work to proceed independent of the register names used for intermediate values. In general, a fully-previsioned renamer requires two read ports and one write port per renamed instruction per cycle, for the case of three-address form instructions. As such, it is a highly-specialized RAM-based structure. Furthermore, renamers usually also maintain a freelist of unbound physical registers. Instructions return registers to the freelist when they commit new architectural values for the associated architectural register. Instructions consume registers from the freelist when they define new values for architectural registers, at rename-time. As a consequence of the current access requirements, neither renamer nor freelist scales well in *W*.

Renaming assumes the use of a Physical Register File (PRF). The number of registers in the PRF needed to prevent rename stalls due to register insufficiency, in the worst case, is $N_{archregs} + N_{flight}$, the sum of the number of renamed architectural registers and the number of instructions in flight (i.e., renamed but non-committed instructions). This bound arises because architectural registers must always have an assigned value (hence $N_{archregs}$), and each in-flight instruction can define a new speculative value for an architectural register (hence N_{flight}). As a result, the number of registers needed in the PRF grows with the window size.

Decode & Rename[5] flows *W* instructions per cycle to the *Dispatch* stage. To this point, instructions have flowed in predicted program order, and *Dispatch* will insert instructions into the out-of-

order components of the pipeline. In particular, at *Dispatch* each instruction is allocated an entry in a Re-Order Buffer (ROB). The contents of each ROB entry are implementation-specific, but its purpose is to recover true program order in the midst of actual out-of-order execution. Enough information about each instruction is stored in the ROB to suit this purpose.

Concurrent to ROB entry allocation, *Dispatch* also inserts instructions into the instruction scheduling hardware (variously called the *issue queue*(s) (IQ), *scheduler*, *scheduling window*, etc.), which is typically smaller in total size than the ROB. The scheduler gives rise to out-of-order execution. By examining data dependences and register availability, the scheduler determines which instructions are ready to execute, independent of the order in which those instructions appear. During *Wakeup & Selection*, instructions with ready source operands are identified, irrespective of program order, and issued to functional pipelines. The scheduler is typically aware of effective pipeline latencies in the datapaths, and schedules instruction issue accordingly. For example, successors of a register value can often issue on the cycle after their predecessors, as execution datapaths usually implement full bypassing.

The scheduler constitutes another critically-constrained structure in the out-of-order microarchitecture. The scheduler must accept W instructions per cycle from *Dispatch* (depending on the scheduler implementation, dispatch into the scheduler may not be contiguous, requiring specialized logic to manage scheduler space), and must select some number of instructions each cycle for issue (typically $W_{issue} \geq W$, to account for differences in instruction type). Scheduling is com-

5. In the out-of-order design used in this study, NoSQ [146] implements its memory disambiguation predictions at the conclusion of *Decode & Rename*. I omit thorough discussion of traditional snooping memory disambiguation, as it is not used in this thesis.

monly accomplished with associative search over register identifiers in a content-addressable memory (CAM) [190], or with a per-physical-register matrix [70, 144].

The functional units (datapaths) themselves constitute the *Execution* stage of the pipeline. At this point, instructions are out of program order. During *Execution*, arithmetic computations are performed, and memory is accessed (i.e., the L1-D is consulted, or a miss is handled by allocating a Miss Status Holding Register (MSHR) and accessing the appropriate element of the memory hierarchy). Physical register tags for input and output operands accompany instructions through the functional pipelines. When the input operand tag of a functional pipe matches an output operand's tag, a bypass network is triggered to forward the result value to the successor instruction as appropriate, to maintain the illusion of program order. Overall, complexity of the bypass network scales quadratically in the number of functional units [68].

After instructions complete execution, they write any results into the PRF during the *Write-Back* (*WB*) pipeline stage. After write-back, instructions reside only in the ROB, until *Commit* determines that a particular instruction is now the eldest such instruction in flight. *Commit* logic updates the architectural program counter (aka, instruction pointer), writes any stored values to memory (or a store buffer), releases physical registers to the freelist, and triggers any latent precise exceptions associated with the committing instruction.

On the whole, the canonical out-of-order core is a product of the era in which it was developed. In several areas, non-scalable structures are employed because, in previous technology generations, energy-efficiency was not an overarching consideration. For instance, schedulers broadcast over large structures, full bypass networks incur N^2 complexity, and register files are

centralized. However, the research and industrial community recognized these obstacles. Many proposals emerged targeting these challenges individually, or on the whole.

2.3 Related Work

Out-of-order design has been refined over the course of decades. Textbooks have been written on the subject. Instead of attempting to cover all aspects of prior work in core microarchitecture, I discuss a pertinent subset of work, often representative of other similar proposals. Briefly, I categorize related work into five principal categories: static scalability (Section 2.3.1), scalable instruction scheduling (Section 2.3.2), optimizations pursuing performance through MLP (Section 2.3.3), distributed microarchitectures (Section 2.3.4), and lastly, dynamic microarchitecture scaling (Section 2.3.5).

2.3.1 Large-Window Machines and Complexity-Effective Designs

There have been many proposals for implementation of large-window machines. *Split Window Processing* [54] sought to construct (effectively) a large instruction window by constructing several smaller windows. Smaller windows limit the inherent delays in searching for independent operations, and limit the scope of some data dependences (e.g., to reduce demand for broadcasts).

Instead of addressing only the individual scalability problems of single cores, *Multiscalar Processing* [152, 53, 55, 118, 174, 24] takes another approach altogether: use many processing units[6] to accelerate a single thread. Broadly speaking, the Multiscalar approach divided an execution into tasks, and each task would execute independently, subject to true data dependences.

6. In the Multiscalar parlance, "cores" in context were not complete processors, but "Processing Units," capable of executing instructions but relying on an external entity for other "core" functionality, like the handling of memory ordering.

Multiscalar raised the bar on exploitation of ILP by challenging latent assumptions. For example, control-independence can be exploited even after mispredictions within, but limited to, an individual task. Moreover, by replicating processing units, Multiscalar demonstrated the utility of multiple, distributed schedulers, thereby bounding scheduler complexity. Lastly, its decoupled handling of memory dependences showed that scalable disambiguation was achievable, with reasonable hardware, even for large-window machines.

Complexity-Effective Superscalar Processors [126] sought to address the individual non-scalable core components, and approached the problem of the statically-scalable core by *clustering* reduced-complexity microarchitectural structures. By leveraging instruction steering, simple in-order scheduling components can take the place of associative-based out-of-order components. Clustering coupled with effective steering also limits the scope over which bypassing networks and register files must operate. However, in clustered designs, broadcast operations are commonly used as a fallback operation, when in-cluster resources cannot satisfy a demand (e.g., an inter-cluster data dependence).

The Multicluster Architecture [49] leveraged clustering with explicitly copy-in/copy-out queues between clusters to implement a statically-scalable core. The same approach was subsequently used in the Core Fusion [86] work for dynamically scalable cores (more details below). Gonzalez et al. propose an asymmetric clustered architecture [62], in which individual elements of the cluster are optimized for a particular workload characteristic. For example, they suggest that individual cluster complexity can be reduced by using dynamically narrower datapaths, which in turn affects the size of bypass networks, as well as power consumption and wakeup latency. The authors later investigate dynamic cluster resizing [63].

Several studies have been made investigating appropriate steering metrics for clusters [30, 20, 141]. In general, cluster steering algorithms seek to make the common case local (i.e., fast) and the uncommon case correct (e.g., via a broadcast). However, because the design in this work focuses on lookahead rather than peak IPC, optimizing instruction issue width is not prioritized, and the demands on steering to produce efficient communication are lessened.

Cherry [107], "Checkpointed Early Resource Recycling", and follow-on work Cherry MP [113] for multiprocessors, re-evaluates the need for all in-flight instructions to retain resources (nominally, physical registers) until commit-time. Early resource recycling speculatively commits values, and recycles stale physical registers, enabling large-window machines to use smaller physical register files. In the uncommon case of a misspeculation, correct state can be restored from a checkpoint. CPR [4, 3] takes a similar approach to Cherry, with an emphasis instruction ordering. CPR requires no ROB in order to ensure in-order retirement. Instead, ordering is maintained through a set of register state checkpoints.

KILO-instruction processing [40, 39] combine these ideas to create ROB-free checkpoint-based instruction windows, with reduced demands on the size of the physical file. *Decoupled* KILO-instruction processing [128] further innovates on designs using conventional hardware, and divides computation into two general categories: those instructions that can be serviced from in-cache data, assigned to a *Cache Processor*, and those that depend on in-memory (out-of-cache) data, assigned to a *Memory Processor*. Each processor is then individually optimized to best suit the need of its assigned instruction type, e.g., by optimizing the cache processor for high execution locality.

2.3.2 Instruction Schedulers

Because many workloads are limited by latency to memory, it is important for high-performance cores—and scalable cores when scaled up—to service as many memory accesses concurrently as possible. However, a non-scalable instruction scheduler limits how much MLP a core can exploit, due to a phenomenon called *IQ (scheduler) clog* [168], in which the scheduler fills with instructions dependent on a long-latency operation (such as a cache miss). Optimizations exist to attack this problem, e.g., by steering dependent instructions into queues [126], moving dependent instructions to a separate buffer [101, 143], and tracking dependences on only one source operand [95]—all of which allow the construction of larger schedulers, less prone to clog. These proposals ameliorate, but do not eliminate, the poor scalability of traditional instruction schedulers.

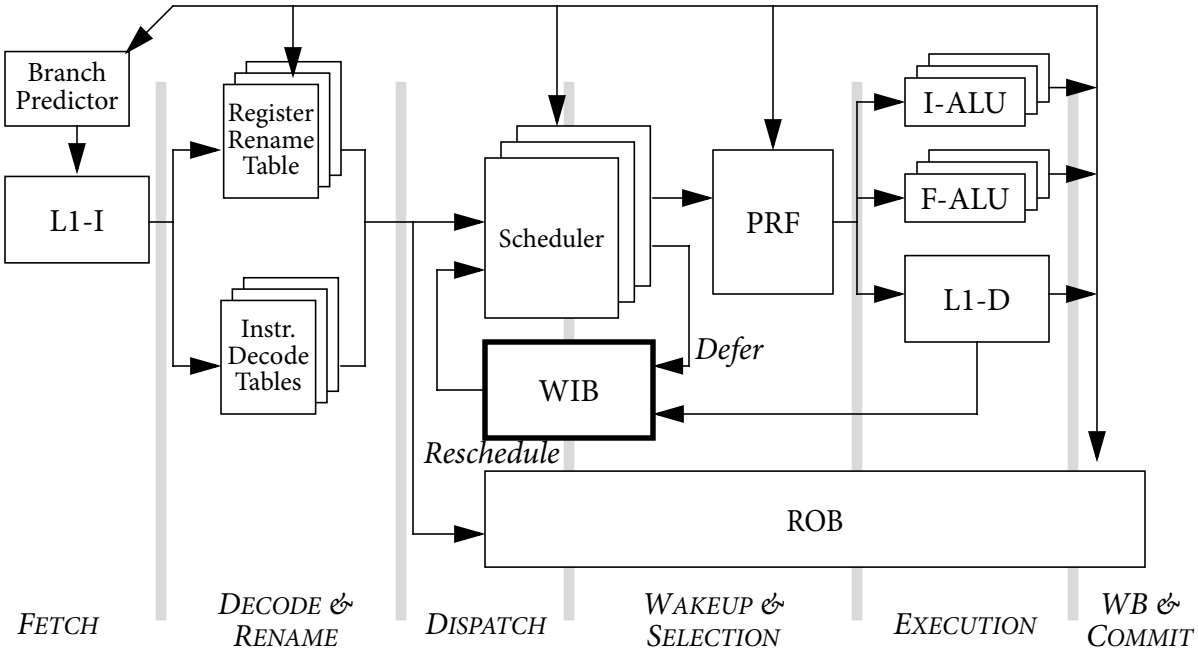


FIGURE 2-6. Out-of-order microarchitecture augmented with a Waiting Instruction Buffer (WIB).

Direct Wakeup [136] and *Hybrid Wakeup* [78] both use a single dependence pointer to designate the first successor of each instruction producing a value, but fall back on a full-scheduler broadcast in the case of multiple successors, with different broadcast implementations. By using a broadcast mechanism as a fallback, these schedulers are still cycle-limited by the worst-case broadcast time. The Matrix Reloaded scheduler [144] re-evaluates the traditional organization of a broadcast-based matrix scheduler by filtering broadcasts of register availability to a handful of entries in a much larger aggregate scheduler.

The Half-Price Architecture [95] attacks the scheduling problem by observing that most operations dynamically wait for only one operand, and implements "half" of a traditional scheduler for most operations. Literally, this leads to a half-as-costly implementation of scheduling logic (e.g., with one half the number of entries in a scheduling CAM).

Some might argue that the *Waiting Instruction Buffer* (WIB) [101] approach is more suited to the next section (exploitation of MLP). Lebeck et al. suggest that IQ clog can be avoided by draining instructions doomed to wait on a long-latency event (e.g., a cache miss) from the scheduler, and storing them in a simple buffer. By draining the scheduler, independent instructions can flow into scheduling logic and be considered for execution. When the long-latency event completes, instructions can be re-dispatched into the scheduler, ready to execute. Figure 2-6 shows the modification necessary to the canonical out-of-order core to include a WIB (bolded).

I include the WIB as a scheduler optimization because the microarchitectural changes needed to incorporate WIBs are largely limited to the scheduler, though a similar design (Continual Flow Pipelines) is discussed below.

2.3.3 Designs Exploiting Memory-Level Parallelism

Two main proposals in the area of MLP-optimized design have sparked a gamut of related projects. Like the WIB, both *Runahead Execution* and *Continual Flow Pipelines* address the scheduler problem by draining a scheduler of non-ready instructions (e.g., by buffering instructions [156] or by simply discarding them [46, 122]).

Continual Flow Pipelines (CFP) [156], and similar designs such as iCFP (in-order CFP) [74] and Sun's ill-fated Rock project [97, 35], combine the observations of checkpoint-based microarchitectures, early resource release, and instruction deferral to further advance the state of the art. Instruction deferral alone exposes MLP, but deferred instructions hold critical processor resources, e.g., physical registers. This places great demands on the PRF organization, as well as the rename logic. However, deferred instructions are guaranteed not to actually make use of the resources held on their behalf—CFP innovates by releasing resources held by deferred instruc-

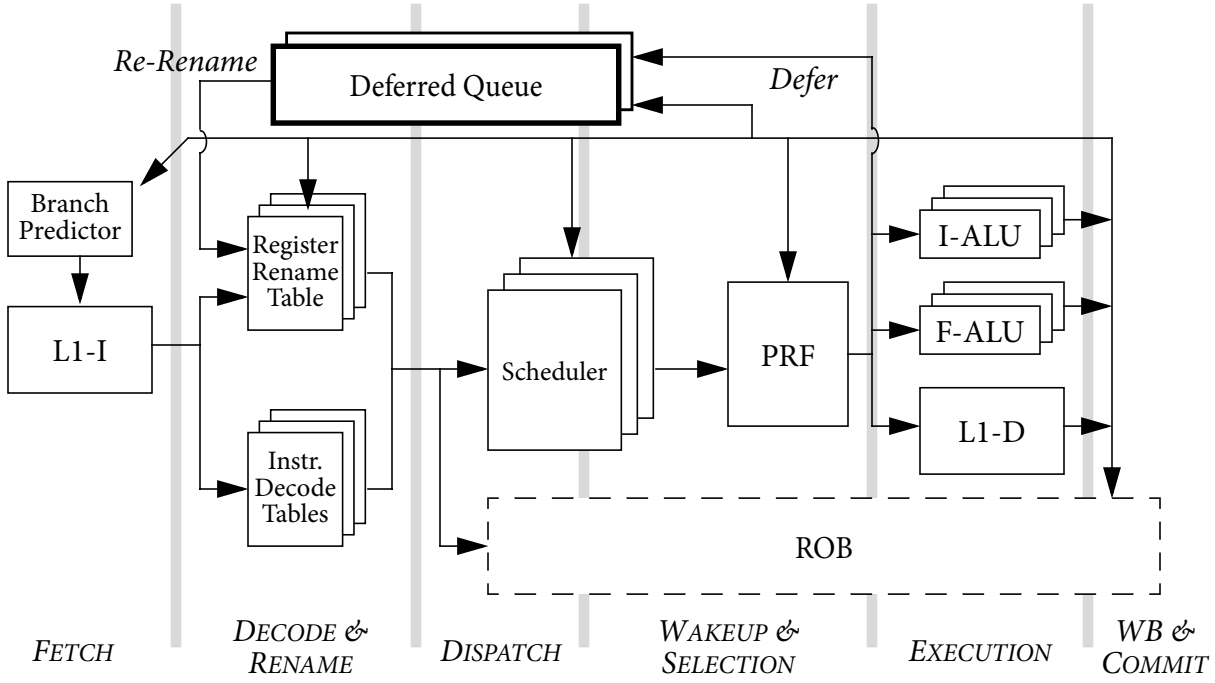


FIGURE 2-7. Out-of-order microarchitecture augmented Continual Flow-style Deferred Queues.

tions, and then re-acquiring them at the end of a deferral period. Figure 2-7 shows the microarchitectural changes[7] necessary to do so: the addition of *Deferred Queues (DQs)*—specialized hardware for retaining deferred instructions, similar to a WIB. Deferred instructions release their physical registers upon deferral, and the underlying hardware can re-use those registers for subsequent (ready) instructions. When leaving a DQ, instructions re-acquire a physical register, and subsequent successors of that register undergo re-renaming to ensure correct dataflow dependences are established.

Like the WIB approach, CFP makes more efficient use of traditional scheduling hardware, by preventing non-ready instructions from clogging the scheduler. However, re-dispatch and re-renaming operations are fairly expensive, and places additional demand on those structures. Worse still, depending on inter-instruction dependences, instructions may defer and subsequently re-rename and re-dispatch several times before they are finally executed. CFPs literally *flow continuously*, i.e., the activity factors on pipeline structures tend to be much higher than a pipeline that stalls on long-latency events (I evaluate CFP quantitatively in Chapter 5), thereby raising power consumption concerns.

7. CFP is based on CPR [4], which uses checkpoints in lieu of a ROB. Hence, I show the ROB outline dotted, to indicate that the CFP proposal uses another mechanism in its place.

Runahead Execution follows a similar vein as CFP, but implements instruction deferral by completely discarding deferred instructions. When a load miss is encountered, a runahead processor enters runahead mode, a form of deep speculation. The goal of runahead is to locate and pre-execute independent memory misses, initiating data movement in a long-latency memory hierarchy well in advance of the time at which a traditional architecture would execute the same instructions. By redundantly pre-executing memory instructions, runahead effectively prefetches data into nearby caches, which can then be used quickly once runahead mode terminates.

During runahead mode, the processor discards operations dependent on long-latency misses, draining them from the scheduler and allowing independent work to proceed. All instructions that "commit" during runahead mode are discarded, and do not update architectural state. Stores executed under runahead store their values into a special-purpose runahead cache, snooped in parallel with the L1-D cache for loads in runahead mode to maintain store-to-load dependences.

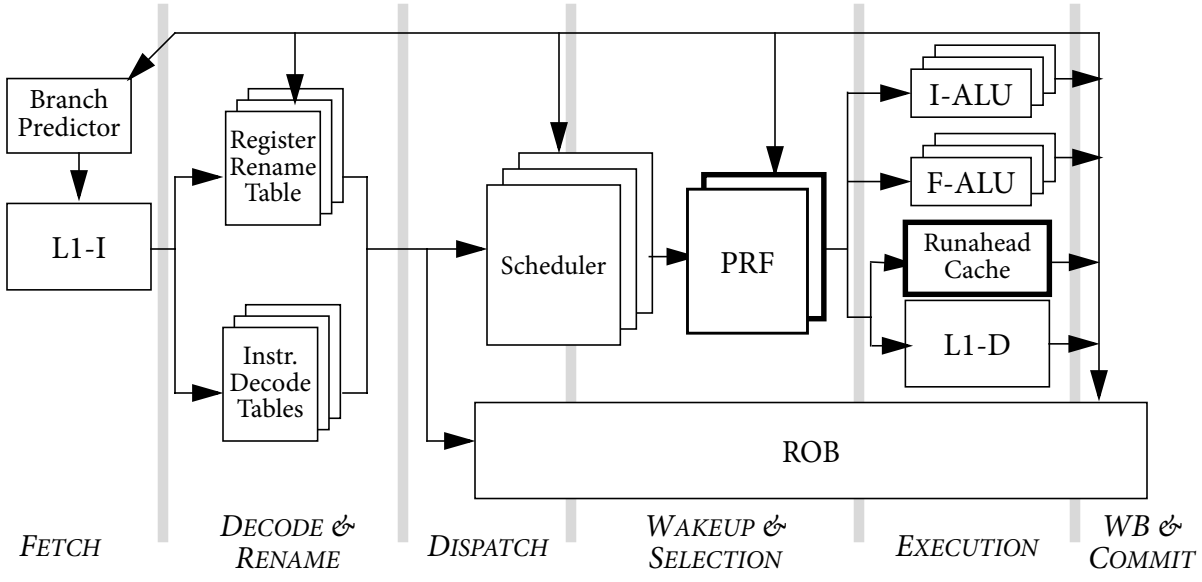


FIGURE 2-8. Out-of-order microarchitecture augmented with Runahead Execution.

Runahead mode ends when the long-latency operation that initiated runahead mode completes. At that time, a checkpoint of the PRF (or a logical checkpoint, using the renamer) is restored (indicated in bold in Figure 2-8), eliminating any updates to register state incurred during runahead mode. The end of runahead mode also clears the runahead cache.

Runahead execution is a popular topic. There have been at least two runahead-related patents [186, 130]. Multu et al. study runahead efficiency [120]. Ramirez et al. proposal runahead for simultaneously-multithreaded cores [137]. Xekalakis et al. advocate runahead helper threads [187], and IBM produced a commercial implementation, the IBM POWER6 [28].

Among CFP and Runahead-like proposals, the critically-constrained resources are, again, the physical register file, and the scheduler. Rather than attempt to make conventional microarchitectural structures more efficient, the work in Chapters 4 and 5 decouples the register file from the number of in-flight instructions, and builds a scheduler in which space is not constrained, by implementing a scalable, distributed scheduler.

2.3.4 Distributed Designs

The TRIPS project [123, 142, 92, 64] represents an ambitions top-to-bottom effort to rethink the organization of a CPU, leveraging a new ISA and distributed, heterogeneous execution tiles. Rare among project in academia, the TRIPS project culminated in a hardware implementation. After recompilation, many single-thread applications show significant speedup. However, the promise of TRIPS-like designs is coupled with the requirement of a fundamentally new execution model, a substantial obstacle to widespread adoption. Nonetheless, the TRIPS architecture's explicit use of forward pointers inspired some of the innovations in this work. So, too, did WaveS-

calar [163], which like TRIPS takes a holistic approach, distributing processing and memory across a chip, but again relies on a novel ISA and compiler support to realize its vision.

The *Composable Lightweight Cores* (CLP) [94] proposal also leverages a novel ISA, but distributes essentially homogeneous simple tiles on a single chip, which can be dynamically composed to produce configurations with better single-thread performance, and dynamically fissioned to revert to a configuration more suitable to large thread counts. CLP represents an extreme of the resource borrowing philosophy, discussed in Section 2.1.2. Core Fusion [86] also composes whole cores, though the individual cores themselves are more substantial than those of CLP.

2.3.5 Dynamic Microarchitectural Scaling

David Albonesi's work on dynamic microarchitectural scaling inspired some of the work in this thesis [8, 18, 27]. Albonesi et al. consider dynamically tuning pipeline width, cache sizes, and instruction queues, to optimize a variety of metrics. This work differs structurally from Albonesi's in that I pursue a modular approach to core scaling, rather than attempt to scale individual structures in unison. Modularity gives rise to more efficient designs, and effectively decouples scaled components from difficult-to-scale resources, like bypassing networks. Huang et al. [81] and Iyer and Marculescu [87] both propose general approaches by which to scale core resources. These (and other) proposals are discussed more fully in Chapter 6, as they constitute baselines for comparison.

2.4 Summary

Scalable cores have the potential to ease the transition from mostly-sequential to mostly-parallel software. However, challenges remain in the design of such cores, and the proper methods for

using them in a chip multiprocessor. In this chapter, I have outlined a vision for scalable cores in the era of multicores, detailed pertinent background material, and discussed the prior work that led to or inspired the work in this dissertation.

The next chapter will detail the methods I use for quantitative evaluation in this dissertation. The subsequent three chapters detail the novel contributions of this work, and Chapter 7 concludes.

Chapter 3

Evaluation Methodology

This chapter details the methods used to evaluate the proposals in this dissertation. I present details of the timing-first simulation infrastructure used in the evaluation, a discussion of power and area estimation, and discuss metrics for energy-efficiency. Following the general infrastructure and methods discussion, I outline salient details of several different machine configurations used in this study. Lastly, I provide a description of benchmarks used in this study.

3.1 Performance Evaluation Methodology

This research focuses on processor core design. This work would not have been possible without a simulation platform capable of simulating all of the different microarchitectures used this work. Therefore, the development of the simulation infrastructure was motivated by the requirement that all target microarchitectures share common underlying design assumptions—especially in the structures not specific to a given target. For example all machines operate with precisely the same model of instruction fetch, branch prediction, etc. The purpose of this homogenization to ensure fair, apples-to-apples quantitative comparison between designs.

During the initial exploration of the research leading to this dissertation, I used trace-based simulation to prototype early designs. While the traced-based framework bore sufficient fidelity to evaluate tradeoffs early in the design process, traces fail to capture the timing-dependent behavior possible in threaded executions [5]. In order to fully capture these effects, the performance evalua-

tions in this thesis used full-system timing-first simulation. In this infrastructure, Virtutech Simics [103, 104, 175], a commercial full-system functional simulator provides a correct execution for the target machine. The remainder of the infrastructure consists of two timing-first simulators: the memory subsystem is modeled using the Ruby memory timing module from Multifacet's GEMS simulator [106, 185], and I and one other student developed a new processor simulator to model the various core microarchitectures used in this dissertation, consisting of approximately 100,000 lines of C++.

Much of the focus of this research is the design of processor core pipelines. Therefore, these pipelines are modeled in great detail. As a result of the level of detail, pipeline stages exhibit significant inter-dependence in their behavior. For instance, between two simulations with slightly different configurations of the instruction window, one might observe a slightly different branch predictor accuracy. Changes in instruction window size affect the timing of branch resolution, which in turn affects predictor training, yielding a slightly different prediction accuracy when window size varies. This example is illustrative—other second-order interdependences exist, as well.

When I report performance results from the simulation, in most cases results are presented as runtime normalized to a baseline design. Runtime is an intuitive metric for both single-threaded and multi-threaded benchmarks, whereas measures like IPC can misrepresent performance for some workloads [6, 7].

3.1.1 Full-System Simulation Implications

Benchmark suites comprised mostly of user-level CPU workloads are widely used in the evaluation of microarchitectures proposed in both academia and industry [172, 146, 87, 86, 79, 168, and

many others]. However, there are several merits in simulating the complete system, including OS interaction and device behavior. Many workload behaviors are only observable via full-system simulation, including system-intensive workloads, which spend substantial time running privileged code. Furthermore, detailed modeling of system activity exposes the performance implications of traps, interrupts, and system calls. Lastly, interactions with devices enables more realistic OS scheduler behavior.

Unfortunately, significant complexity arises in the implementation of timing-accurate simulation of full systems. This complexity is not manageable by single researchers. Timing-first simulation addresses some of this complexity, by modeling only common-case behavior. Uncommon cases are left unimplemented (or implemented in a manner that does not cover all corner cases), and the always-correct functional model is used to re-bootstrap the simulation should the timing-first simulation err. Instead of affecting the correctness of the simulation, these errors manifest as timing transients in the reported performance. I monitor the frequency of this class of errors closely, to ensure these transients do not unexpectedly influence performance results.

Occasional interference from the target's operating system is another consequence of full-system simulation. In particular, though only one process is actively running on the target in most cases, many other processes exist in sleep states, and these processes will occasionally wake and perform background tasks. The target's operating system will also service interrupts at unpredictable and unexpected points in the simulation. These events reflect the operation of real systems, but add noise to data attributed to a given workload. When viewing results for a given benchmark, the reader should be aware that those results are obtained on a machine running not only the objective workload, but also actively servicing interrupts and other background processes in the

manner of actual hardware. To the extent possible, this behavior has been minimized by killing background processes on the target prior to simulation. However, not all interference justly falls into the category of noise. Some system-level behaviors are intentionally invoked by the objective workload (e.g., sending packets on a simulated network), and others occur as implicit side effects of execution (e.g., software-handled TLB fill).

3.2 Power Evaluation Methodology

Power consumption is modeled using the methodology of Wattch [25], with contributions from CACTI [149] and Orion [179] (also based on Wattch). Brooks et al. validated Wattch versus the reported power consumption of a contemporary out-of-order superscalar core, and found that Wattch's power estimates deviate approximately 10% from the predicted estimates from layout-level power estimation tools at the 0.35um feature size.

The basic premise of Wattch assigns a fixed energy cost to microarchitectural events (e.g., a read from a physical register file, or a write into a re-order buffer). In other words:

$$E_{execution} = \sum_{i \in A} Activations_i \cdot E_i \tag{3.1}$$

where A is the set of all microarchitectural components, $Activations_i$ represents the number of activations of each component i in an execution, E_i is the energy consumed by each activation of component i, and $E_{execution}$ is the total energy consumed during the execution. The energy cost of each microarchitectural event (i.e., E_i) is derived from other tools (e.g., CACTI [184, 165, 149]), or from first principles of CMOS logic [e.g., 133].

Wattch models four different circuit types:

- Array Structures: Caches, register files, tables, and other non-associative memories.
- Associative Memories / Content-Addressable Memories: TLBs, LSQs, schedulers, and other associative structures.
- Combinational logic: ALUs, selection logic, comparators, and other combinational components, and
- Clocking: Clock routing, buffering, and loads.

Event types and counts are recorded in the course of simulation, yielding a total energy cost, from which power consumption can be derived. Wattch also models conditional clocking (i.e., clock-gating) by assigning a configurable portion of the nominal energy cost when the unit is under-utilized or not used at all. Unfortunately, the available Wattch implementation models a technology node (0.35um) that no longer reflects reasonable modern power considerations in state-of-the-art CMOS (e.g., 32nm), and does so by scaling linearly from an even older node (0.8um). Therefore, I adopted the premise of Wattch, but with new implementation details in more recent technologies (90nm, 65nm, 45nm, and 32nm).

CACTI [184, 165, 149], a memory layout and power design tool, has been actively developed and improved since its initial release (unlike Wattch). In keeping with the original methodology of Wattch, I elected to use CACTI as a baseline from which to derive estimates of activation energy and leakage power. CACTI was modified to output a more detailed breakdown of energy estimates of individual components (e.g., decoders, arrays, tags, etc.) rather than aggregate estimations. In this manner, it is possible to use CACTI to estimate power consumed by simple arrays, associative structures, and some logic structures, within a common set of design assumptions.

TABLE 3-1. Summary of Power Models (continued)

Structure	Circuit Type	Methodology
BTB	Tagged RAM	CACTI
Bypassing Networks	Logic/Wire	Wattch
Checkpoint Arrays	RAM	CACTI
Clock Distribution and Generation	Logic/Wire	Wattch
Deferred Queue [156]	RAM	CACTI
DQ Bank Crossbar	Wire	Wattch
DQ Bank Group Crossbar	Wire	Wattch
DQ Operand Arrays	RAM	CACTI
DQ Pointer Arrays	RAM	CACTI
DQ Update Logic	Logic	Wattch/Zlatanovici et al. [192]
DQ Valid/Empty-Full BIts	Logic, Flip-flops	Wattch/Markovic et al. [105]
D-TLB	Tagged RAM	CACTI
Flip Flop	Logic	Wattch/Markovic et al. [105]
Floating Point Datapaths	Logic	Oh et al. [125]
I-TLB	Tagged RAM	CACTI
Integer Datapaths	Logic	Matthew et al. [109]
L1-D	Cache	CACTI
L1-I	Cache	CACTI
L2	Cache	CACTI
L3 (Bank)	Cache	CACTI
Microcode Tables	RAM (PROM)	CACTI
Network Links	Wire	Orion [179]
NoSQ	Tagged RAM (Predictor and Store Sequence Bloom Filter); Logic, Flip-flops (Store vulnerability filter)	CACTI; Wattch/Markovic et al. [105]
Parallel Prefix Decoding	Logic	Wattch
Queue Head/Tail Logic	Logic	Wattch/Zlatanovici et al. [192]
Random Logic	Logic	Zlatanovici et al. [192]

TABLE 3-1. Summary of Power Models (continued)

Structure	Circuit Type	Methodology
Register Consumer Table (RCT)	RAM	CACTI
Register Files	RAM	CACTI
Register Freelist	Logic, Flip-flops	Wattch/Markovic et al. [105]
Register Renamer	RAM	CACTI
Re-Order Buffer (ROB)	RAM	CACTI
Return Address Stack (RAS)	RAM	CACTI
Runahead Cache [122]	Cache	CACTI
Scheduler (Hybrid Wakeup) [78]	RAM (Pointer Array); CAM (Fall-back Broadcast Network)	CACTI; Wattch
Slice Rename Filter [156]	Tagged RAM	CACTI
YAGS Branch Predictor	Tagged RAMs	CACTI

Unfortunately, not all elements of the original Wattch model can be derived entirely from CACTI. Selection logic, high-performance CAMs, and other core-specific circuitry do not appear in any design produced by CACTI. For some structures, I made use of the original Wattch formulae, after modernizing technology-specific constants from newer technology generations. Estimates for energy consumption of functional units are derived from published designs implementing the appropriate functionality [125, 109]. Estimates for random logic are derived from the carry-lookahead adder presented by Zlatanovici et al. [192], assumed to have similar energy per area. Pipeline latches are modeled to be similar to those presented by Markovic et al. [105].Table 3-1 summarizes the power models used in this work, the type of circuit modeled, and the methodology used to determine activation energy for each.

The power modeling infrastructure uses the most aggressive clock-gating model from the original Wattch proposal. This model assumes that under-utilized logic consumes energy in linear proportion with its utilization. Hence, a three-ported register file, on which two ports are active in a particular cycle, is assumed to consume two-thirds of its peak dynamic power during the observed cycle.

A collaborative effort with Yasuko Watanabe was made to quantitatively verify the power estimates produced by the above power models. This evaluation was performed with respect to three commercial products: Sun's Niagara 2 [124], Intel's Atom [58], and Intel's Xeon (Tulsa) [164]. Nawathe et al. provide a detailed breakdown of power consumed by individual processor components in the Niagara 2 core when operating at full speed. We compared this breakdown with the data obtained from a simulation of a similar microarchitecture, accounting for differences in core count and functional units. Unfortunately, Gerosa et al. (Atom) and Tam et al. (Tulsa) do not provide similar breakdowns at the level of individual microarchitectural components, however, we were able to validate end-to-end core power consumption.

Not all power models were included in the validation studies. At the time of this writing, no existing products are available as comparison points for the proposed microarchitectures or for some power models used only by non-traditional pipeline designs (e.g., Forwardflow's Dataflow Queue, the slice buffer used by the CFP model, etc.). However, these models were derived with the same methodology used by the verified models, and we expect them exhibit a similar degree of fidelity.

3.3 Energy-Efficiency

This dissertation uses $Energy \cdot Delay = E \cdot D$ and $Energy \cdot Delay^2 = E \cdot D^2$ as measures of energy efficiency. Minimizing $E \cdot D^2$ is equivalent to maximization of $BIPS^3/W$ [1] for a single-threaded workload, as revealed by simple dimensional analysis in Equation 3.2. Assume I is the

1. $BIPS^3/W$ is ratio of the cube of instruction throughput (billions of instructions per second) and power. It is commonly used in publications from IBM.

instruction count of some objective execution, *D* is the time required to complete the execution on the target machine, and *E* is the energy consumed during execution.

$$\frac{BIPS^3}{W} \sim \frac{\frac{I^3}{D^3}}{\frac{E}{D}} = \frac{\frac{1}{D^2}}{E} = \frac{1}{E \cdot D^2} \tag{3.2}$$

Hartstein and Puzak [67] study $BIPS^2/W$ and $BIPS^3/W$ with an analytical model of optimal pipeline depth, and observe that the choice $BIPS/W$ (E/I) as metric argues for a pipeline depth less than one (i.e., a design without any pipelining). This finding also holds for $BIPS^2/W$ when static power contributions are negligible, but that is not the case in this study. Since this research addresses high-performance, and hence, pipelined, processor designs, $BIPS^2/W$ and $BIPS^3/W$ (more generally, $E \cdot D$ and $E \cdot D^2$, respectively) are the logical choices to compare efficiency of designs in this research. I expect Hartstein and Puzak's analysis to hold in this area, as comparison of different microarchitectures will vary similar parameters as those that appear in [67][2].

Srinivasan et al. [157] point out that $BIPS^3/W$ is also an intuitive choice of metric, because it follows the same energy/performance curve as DVFS (dynamic voltage and frequency scaling). They observe that, to first order, DVFS trades off performance in linear proportion for a cubic delta in (dynamic) power. That is, since dynamic power in CMOS $P_D \sim f \cdot V^2$, and $f \sim V$ when operating under DVFS, $BIPS^3/W$ becomes a property of the underlying circuit, not the current

2. E.g., microarchitectural comparisons vary effective pipeline length *p*, total logic delay of the processor t_p, the number of pipeline hazards N_H, the average degree of superscalar processing α, the weighted average of the fraction of the pipeline stalled by hazards γ, and of course, the runtime, *T*., all of which are variables of the Hartstein/Puzak models.

voltage and frequency settings. This argument applies directly to $E \cdot D^2$, of course. However, for completeness, I also include a discussion of $E \cdot D$.

3.4 Area Evaluation Methodology

Area estimates for proposed designs are derived via a combination of automated and manual floorplanning. An automated heuristic-driven genetic floorplanner [140] was developed to provide estimates of area and communication latency from area estimates and qualitative netlists of primitive components. The process was bootstrapped from area estimates produced by CACTI, or from published designs [109, 192], as in Section 3.2.

The floorplanner itself explores multiple frontiers of slicing-tree-based floorplans using a genetic hill-climbing algorithm. The slicing-tree approach can represent horizontal and vertical juxtaposition of adjacent elements, and each element is assumed to rotate arbitrarily in increments of ninety degrees, and to mirror arbitrarily. The floorplanning itself conservatively over-assigns individual unit area, preserving aspect ratio, and uses a Manhattan distance heuristic to estimate interconnect fitness. The methodology assumes that unused area can be partially occupied by unmodeled control logic (assumed to be small relative to major components) or can be used as buffer capacitance for the on-chip power supply. Fitness criteria heavily weight area minimization over routing fitness (elements involved in floorplanning are implicitly assumed to be somewhat insensitive to inter-element communication latency).

The automated floorplanning process was applied hierarchically on progressively larger input blocks, with manual partitioning of input nodes at each stage. The partitioning process itself was ad-hoc; primitive and aggregate units were selected for floorplanning in the same stage when they had (subjectively) similar area and were likely to share common nets, to avoid long searches of designs with little practical impact on overall design area.

The topmost two levels of the floorplanning hierarchy, core and chip floorplans, were completed with manual floorplanning. The principal reason to prefer manual floorplanning at these

levels was to ensure that tiles (consisting of a core, an L2 cache, and a bank of a shared L3 cache—see Section 3.6.2) could be easily replicated and mirrored to build a CMP. To ease the manual floorplanning process, the final stage of automatic floorplanning was augmented to prefer a particular aspect ratio, even at the expense of area efficiency, for selected circuit partitions.

3.5 Target Architecture Model

In the timing-first simulator, the functional authority (Virtutech Simics) fully models an UltraSPARC-III+-based [75] server. However, it is desirable for microarchitectural research to be broadly applicable to a variety of ISAs and architectures. Therefore, I evaluate SPARCv9 [181] with optimizations to SPARC-specific TLB fill and register windowing, motivated by a closer resemblance to the behavior of commodity x86 and x64 ISAs. In particular, the infrastructure models hardware extensions for common-case TLB fill (normally handled with a fast trap in SPARCv9), and speculative inlining of many register window exceptions. These extensions help to decouple the results in this dissertation from the SPARCv9 ISA and its successors, and also improve performance of the target machines. Appendix A quantitatively evaluates the effect of these optimizations on some of the target machines used in this work.

All targets share a common memory consistency model: Sequential Consistency (SC) [100, 72]. Four arguments support the use of SC in this research. First, SC is extremely restrictive of the legal ordering of memory operations with respect to program order (i.e., SC allows no visible reorderings). It is desirable that the microarchitectures studied in this research be widely applicable, and by ensuring that no proposal violates the most restrictive memory consistency model, it follows that all weaker models consistency models are also respected.

Secondly, SC greatly simplifies debugging the simulation itself. Third, the functional authority (Simics) abstracts each CPU with the capability to single-step through an instruction stream. This abstraction effectively forces the functional authority to observe an SC execution. Even if the timing target observed a weaker execution (e.g., a forwarded value from a store buffer, as in TSO), it would violate the correctness criteria of the infrastructure to force the same execution on the functional authority.

Lastly, NoSQ [146] is used throughout all target models to disambiguate multiple in-flight memory references. While NoSQ is speculative in nature, and requires load replay when vulnerable loads are committed. New, original research would be required to adapt NoSQ to weaker memory models, and since this dissertation focuses on the microarchitecture of the instruction window, I leave that work for others. Instead, each target aims to deliver an aggressive SC implementation [61].

3.6 Target Microarchitectural Machine Models

Several instruction window designs are evaluated in this dissertation. These microarchitectures can be broadly dichotomized as either *realistic* or *idealized* models:

- *Realistic Models* represent a complete timing abstraction for the given target. The effects of bank interleaving, wire delay, arbitration, contention, and buffering are fully modeled, as they would be in hardware. Power is modeled with equal rigor. All designs proposed in this research are classified as *Realistic Models*.

- *Idealized Models* leverage simplified abstractions of functionality that would require original research to bring to full fruition as a realistic model, or have already been explored in prior work, and the details have been simplified for ease of implementation. Idealized models are employed as baselines for comparison, and as thought experiments, and are not usually paired with an accurate power model.

3.6.1 Common Microarchitectural Structures

Many target models share common infrastructure in the areas of the pipeline not directly affected by the microarchitecture of the instruction window. These commonalities are summarized in Table 3-2.

TABLE 3-2. Common Configuration Parameters

Component	Configuration
Branch Predictor	YAGS [47], 4K Prediction History Table, 3K Exception Table, 2KB BTB, 16-entry RAS
Disambiguation	NoSQ [146], 1024-entry predictor, 1024-entry double-buffered store sequence bloom filter (SSBF)
Fetch-Dispatch Time	7 Cycles, plus I-Cache miss if applicable
L1-I Cache	32KB, 4-way, 64B line, 4-cycle, pipelined, 2 lines per cycle, 2 processor-side ports
L1-D Cache	32KB, 4-way, 64B line, 4-cycle LTU, write-through, write-invalidate, included by L2, parity, private
L2 Cache	1MB, 8-way, 4 banks, 64B line, 11 cycle latency, write-back, SECDED ECC, private
L3 Cache	8MB, 16-way, 8 banks, 64B line, 24 cycle latency, SECDED ECC, shared
Main Memory	2 QPI-like links [91] (Up to 64 GB/s), 300 cycle mean latency (uncontended)
Coherence	MOESI-based Directory Protocol
On-Chip Interconnect	2D Mesh, 16B bidirectional links, one transfer per cycle, 1-cycle 5-ary routers, 5 virtual channels per link
Consistency Model	Sequential consistency
Store Issue Policy	Stores issue permissions prefetch at Execute
Feature Size	32nm
Clock Frequency	3.0 GHz

All microarchitectural models operate on instructions consisting of 3-tuples. Each tuple consists of up to two input operands (registers or immediate values), and up to one output operand (typically, a register). Not all instructions in the target ISA (SPARCv9) meet this abstraction, and a common layer of micro-operation decoding decomposes instructions that exceed this abstraction into a stream of multiple, 3-tuple instructions. Each micro-operation is allowed to dispatch, issue, and execute independently, and all window models must honor dataflow dependences micro-operations as though they were regular instructions. Two additional logical registers are added for use solely by the instruction decode hardware, for this purpose. The decomposition circuitry precedes model-specific decoding operations in the decode pipeline (e.g., register renaming).

All target models use a NoSQ-like [146] memory disambiguation scheme (subsequently, "NoSQ," referring to the general technique). As proposed by Sha et al., NoSQ (subsequently, "Canonical NoSQ," referring to the specific implementation suggested by Sha et al.) leverages speculative memory bypassing, and identifies loads likely to forward from earlier stores in program order. This speculation is verified through a combination of store vulnerability filtering [139] and commit-time load replay. Misspeculations are used to train predictive structures, and are resolved with pipeline flushes. Canonical NoSQ anticipates DEF-STORE-LOAD-USE chains, and manipulates register rename tables to short-circuit these chains into DEF-USE chains. Sha et al. show that Canonical NoSQ can outperform reasonably-sized associative store queues.

NoSQ is well-suited to large window designs for several reasons. First, it can handle all in-flight instructions in a large window without the need for any fully-associative structures, which have limited scalability. Second, when properly warmed, NoSQ's predictors exhibit very low false positive (improper predicted store-to-load dependence) and false negative (unanticipated store-

to-load dependence) rates. Third, NoSQ can easily handle cases in which a single store bypasses to several dependent loads (though it does not handle the case of loads bypassing from multiple elder stores). Unfortunately, Canonical NoSQ is deeply tied to a traditional out-of-order pipeline, as it relies on register renaming. The more general model of NoSQ employed in this work instead uses mechanisms specific to each target model to represent (predicted) store-to-load dependences. The precise mechanism used is described in subsequent target-specific sections.

Furthermore, the correctness-checking mechanism in the infrastructure does not easily allow outright cancellation of loads and stores, as advocated in Canonical NoSQ. Hence, the NoSQ model leveraged in the simulation allows loads predicted to bypass to flow through the pipeline normally. These loads are subject to injected dependences that prevent the loads from issuing until their forwarding store completes execution.

3.6.2 CMP Organization

This thesis evaluates a hypothetical 8-core CMP. Singe-threaded evaluation is carried out as though only one core of this target CMP was active: other cores are assumed to be in a halted state (or off), in which the core and private caches consume no power and perform no work. Contemporary CMPs already use a similar paradigm to conserve sufficient energy in order to run some cores at increased speed via DVFS [85].

Figure 3-1 depicts this dissertation's 8-way CMP target machine. On each tile resides a single core, a private L1-I cache (32KB), a private write-through write-invalidate L1-D cache (32KB), a private L2 cache (1MB) which manages coherency in the L1-D via inclusion, and one bank of a shared L3 cache. It is assumed that cores and private caches can be powered off without affecting the shared L3—i.e., the L3 operates in its own voltage domain. The interconnect between tiles is a

2D mesh, which supports a 5-state MOESI directory-based coherence protocol. Two memory controllers provide 32 GB/s of bandwidth to DRAM (each).

The figure expands the core-private resources (left), indicating the detailed floorplan of a core in the specific case of a Forwardflow core, provisioned with four bank groups. Though individual core microarchitectures vary somewhat in area, the performance model assumes that each target core's area and aspect ratio are not substantially different to warrant a different CMP floorplan, nor does area/aspect ratio differences affect timing of common structures.

Subsequent subsections detail each target CPU class: *RUU* (Section 3.6.3), *OoO* (Section 3.6.4), *Runahead* (Section 3.6.5), *CFP* (Section 3.6.6), and *OoO-SSR* (Section 3.6.7).

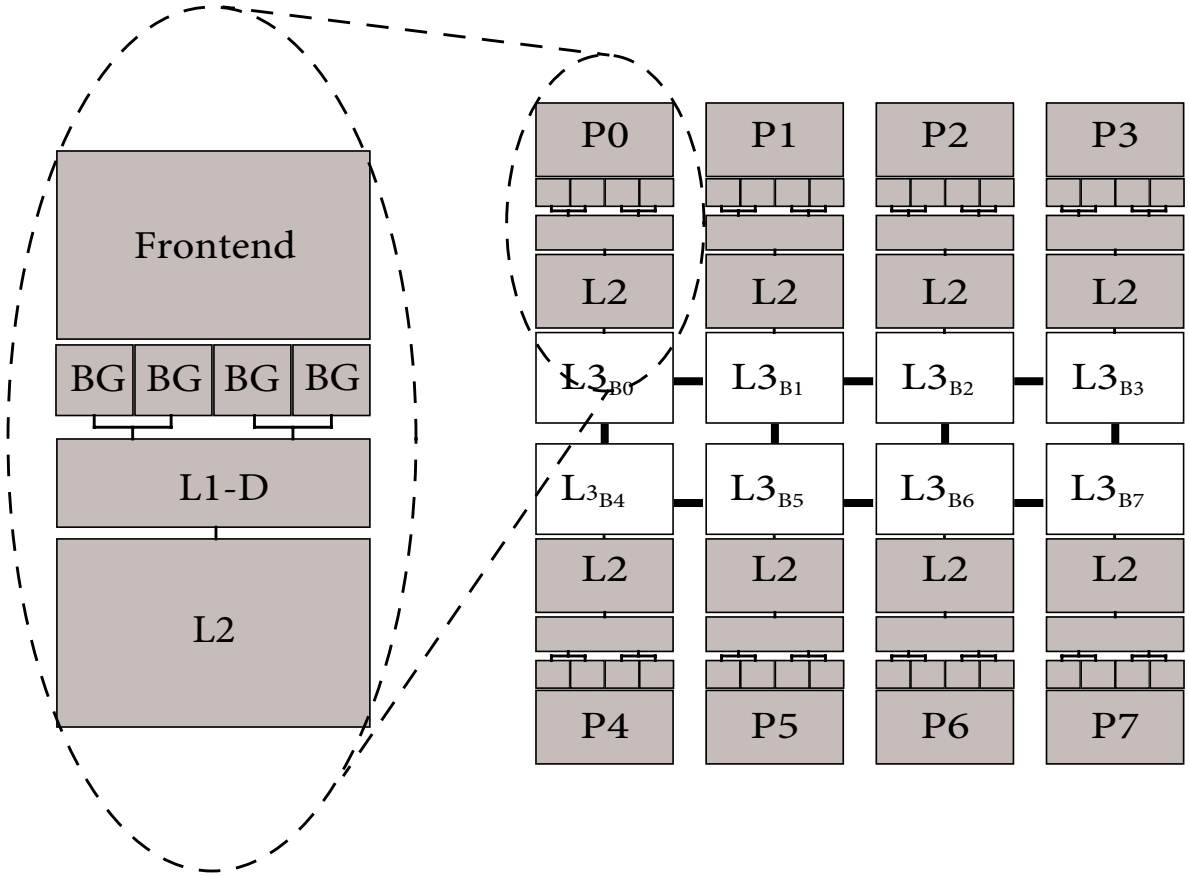


FIGURE 3-1. 8-Way CMP Target

3.6.3 *RUU*

The *RUU* machine configuration is an idealized model, representing an approximate upper bound on performance for a given window size. The overall operation of *RUU* is similar to the design described by Sohi and Vajapeyam [154, 153]. This baseline is idealized in several ways:

- All *RUU* configurations are able to schedule any instruction in their window, regardless of window size. That is, *RUU* never suffers from scheduler clog [168].
- All *RUU* configurations are amply provisioned with functional units, up to the maximum used by any other configuration of the same window size (the largest configuration, *RUU-1024*, includes 40 functional pipelines).
- All *RUU* configurations' register files are provisioned to never cause stall conditions (i.e., read/write port provisioning). *RUU* uses an architectural register file (ARF).

RUU approximately represents the maximum attainable performance for a fixed window size, frontend configuration, and ALU budget. *RUU* is an *approximate* upper bound because its scheduling does not account for instruction criticality [169]. While not the focus of this work, some other designs can coincidentally exploit instruction criticality at scheduling-time.

Because of the idealized nature of the *RUU* model, power estimates for *RUU* are not provided.

3.6.4 *OoO*

The *OoO* target represents a contemporary out-of-order superscalar processor core, typified by the Alpha 21364 (EV7) [88, 41]. *OoO* is a realistic model, in that most microarchitectural structures are constrained to reasonable sizes, are subject to wire delay, and have constrained bandwidth.

OoO uses the scalable CAM-based scheduler described by Huang, Renau, and Torrellas, *Hybrid Wakeup* [78]. This scheduler uses a forward pointer to designate the first successor of an operand, and resorts to a broadcast-based mechanism in the case of multiple successors. The common case of a singleton successor is handled in this scheduler without a broadcast, reducing its power consumption. However, because this scheduler relies on a CAM as a fallback mechanism, it is assumed in this dissertation that *OoO*'s scheduler does not scale to full-window size.

OoO uses a centralized, renamed physical register file (PRF). The PRF contains sufficient registers such that a stall condition never arises. Furthermore, ports on the PRF are fully provisioned to allow uninterrupted issue and writeback operations. To handle writeback races, all read ports in the PRF are fully bypassed to all write ports. Fully provisioning both capacity and port count yields a large PRF design, which is assumed to require only a single cycle to access. This latency is feasible in a hierarchical register file design (e.g., the register file used in the UltraSPARC III [98]), though internal details of this register file are not modeled.

3.6.5 *Runahead*

The *Runahead* model is a realistic model, based on *OoO*, that implements Runahead Execution [46, 122]. Runahead Execution attempts to expose additional memory-level parallelism when servicing cache misses by checkpointing processor state and entering a speculative (runahead) mode. In runahead mode, a core discards instructions dependant on long-latency operations, such as a cache miss, and continues execution well ahead of the current commit point. When the long-latency operation that triggered runahead mode is resolved, processor state is restored from a checkpoint and execution resumes.

The principal benefit of runahead execution is its ability to prefetch cache lines that will be accessed in the near future by the target instruction stream. Though all work in runahead mode is effectively discarded, merely by issuing requests to the memory hierarchy, *Runahead* is able to generate useful prefetches for some workloads, enabling those access to hit in a nearby cache when they are re-executed non-speculatively. However, the utility of runahead mode is strongly dependent on workload.

Runahead mode comes at a disadvantage of discarding potentially useful work, potentially in the course of many distinct invocations of runahead mode. In particular, because uncommitted values are squashed without retirement in runahead mode, completed data-independent instructions that follow the long-latency triggering event must be redundantly executed, requiring additional execution energy, and also incurring opportunity cost in the form of additional functional unit utilization and a repeat traversal of the frontend structures by the redundantly-executed instructions.

Runahead is a variant of *OoO*. To enable runahead mode, *OoO*'s physical register file is augmented with poison bits, to identify values not available in runahead mode. Initially, only the register designated as output from the initiating load is poisoned. Poison bits propagate to other registers when, instead of executing, instructions with one or more poisoned operands skip execution, poison their output register, and are discarded. Furthermore, to accommodate additional registers constituting the checkpoint of architectural state need at the end of runahead mode, the PRF is augmented with additional registers.

Lastly, *Runahead* is augmented with a runahead cache [122], which is used to buffer the values of speculative stores while in runahead mode. The runahead cache is flash-cleared at the end of

runahead mode, is filled by runahead stores, and is snooped to service every load instruction in runahead mode.

3.6.6 Continual Flow Pipeline (*CFP*)

The *CFP* model is a partially idealized model based on the *Continual Flow Pipeline* design proposed by Srinivasan et al. [156] and on Sun's Rock [35]. Like Runahead Execution, *CFP* pursues increased memory-level parallelism, but instead of discarding instructions dependant on long-latency operation, *CFP* defers instructions into scalable buffers, called *Deferred Queues* (DQs).

In a Continual Flow Pipeline, the frontend should never stall due to IQ clog or similar phenomenon. Whenever the scheduler is filled, the oldest outstanding non-deferred load operation is marked as deferred, its output register is poisoned, and instructions dependant on that register are drained from the scheduling logic into an available Deferred Queue. When an instruction enters a DQ, it releases its physical registers for immediate re-use by other instructions. Instructions in Deferred Queues wait until the load that caused their initial deferral completes execution, and then re-enter the pipeline at the Rename stage, re-Renaming and re-acquiring physical registers as needed.

CFP is a compelling design because it conserves valuable space in scheduling logic by buffering instructions that would normally clog the scheduler in easily-scalable SRAM-based DQs. However, once a DQ begins to drain it must drain completely, lest the scheduler re-fill with instructions that occur *later* in the predicted instruction stream than the oldest non-deferred load. Should such a condition arise, subsequent deferral of outstanding load operations is not sufficient to guarantee that the scheduler will ever issue another instruction, thereby risking deadlock. Therefore, though CFP will never re-execute an instruction, it may necessarily (and by design) re-

defer an instruction many times before it is finally executed, incurring energy overhead on each such occasion.

The *CFP* model is based on *OoO*. In addition to the structures present in *OoO*, all *CFP* models are augmented with one or more Deferred Queues and accompanying deferral logic, including poison bits in the register file and logic to associate deferred slices with register live-outs. Because physical registers are not held by deferred instructions, *CFP* requires no additional space in the physical register file. However, to implement re-Renaming and re-dispatch, *CFP* utilizes a renaming filter, as described by Srinivasan et al., which facilitates register re-acquisition as DQs are emptied.

Unlike CFP, which uses Checkpoint Processing and Recovery (CPR) [3] to implement commit sequencing, *CFP* (i.e., the model) uses a traditional re-order buffer to maintain correct program order for all instructions. This is a departure from Srinivasan et al., however, unlike schedulers, ROBs are easily scaled to large sizes, such that *CFP* is not often starved for window space.

Because the ROB effectively bounds the number of in-flight instructions in the *CFP* model, the DQs themselves have unbounded capacity. This idealizes *CFP* somewhat, as unbounded DQs allow any or all available DQs to grow to arbitrary size (less than or equal to that of the ROB) without penalty. This idealization simplifies the deferral-time steering logic, which selects the DQ to which an instruction will be deferred, as DQ resource constraints do not interfere with optimal dataflow-based instruction steering.

3.6.7 *OoO-SSR*

OoO-SSR (Out of Order, Serialized Successor Representation—see Chapter 4) is an idealized model of a traditional out-of-order core that uses a scheduler similar to the pointer-based schedul-

ing of Forwardflow. Though nearly all aspect of *OoO-SSR* are realistic, its scheduler is idealized to behave identically to the scheduling of a Forwardflow core. *OoO-SSR* is useful in understanding the performance impact of serialized wakeup in particular, and SSR in general, as compared to the performance achieve by an *RUU* configuration of identical size.

3.7 Benchmarks

The evaluation in this thesis uses a variety of single-threaded and multi-threaded workloads. SPEC CPU 2006 [69] constitutes the majority of the single-threaded benchmarks. A complete issue of *Computer Architecture News*[3] is devoted to analysis of SPEC CPU 2006. I also include single-threaded versions of the Wisconsin Commercial Workload Suite [5]. Descriptions of each of these workloads are provided in Table 3-3.

TABLE 3-3. Single-Threaded Workload Descriptions (continued)

Benchmark	Suite	Language	Description
`astar`	SPEC INT 2006	C++	AI: Pathfinding
`bzip2`	SPEC INT 2006	C	Burrows-Wheeler transform compression, combined input set.
`gcc`	SPEC INT 2006	C	C-language optimizing compiler. 200.i input.
`gobmk`	SPEC INT 2006	C	AI: Game of Go
`h264ref`	SPEC INT 2006	C	Video compression codec
`hmmer`	SPEC INT 2006	C	Gene sequence search
`libquantum`	SPEC INT 2006	C	Quantum computer simulation
`mcf`	SPEC INT 2006	C	Combinatorial optimization
`omnetpp`	SPEC INT 2006	C++	Discrete event simulation
`perlbench`	SPEC INT 2006	C	Perl v5.8.7, running SpamAssassin
`sjeng`	SPEC INT 2006	C	AI: Chess
`xalancbmk`	SPEC INT 2006	C++	XML to HTML transformation
`bwaves`	SPEC FP 2006	F77	Computational fluid dynamics, 3d transonic transient laminar viscous flows

3. *Computer Architecture News* Vol. 35, No. 1 - March 2007

TABLE 3-3. Single-Threaded Workload Descriptions (continued)

Benchmark	Suite	Language	Description
cactusADM	SPEC FP 2006	C, F90	Numerical relativity solver
calculix	SPEC FP 2006	C, F90	Structural mechanics
dealII	SPEC FP 2006	C++	Finite element estimation
gamess	SPEC FP 2006	F90	Quantum chemical computations
GemsFDTD	SPEC FP 2006	F90	Computational electromagnetics
gromacs	SPEC FP 2006	C, F	Molecular dynamics
lbm	SPEC FP 2006	C	Computational fluid dynamics, Lattice-Boltzmann method
leslie3d	SPEC FP 2006	F90	Computational fluid dynamics, finite-volume MacCormack Predictor-Corrector
milc	SPEC FP 2006	C	Quantum chromodynamics
namd	SPEC FP 2006	C++	Structural biology simulation
povray	SPEC FP 2006	C++	Ray-tracing
soplex	SPEC FP 2006	C++	Linear program solver
sphinx3	SPEC FP 2006	C	Speech recognition
tonto	SPEC FP 2006	F95	Quantum crystallography
wrf	SPEC FP 2006	C, F90	Weather forecast modeling
zeusmp	SPEC FP 2006	F77	Magnetohydrodynamics
apache-1	Wisconsin Commercial Workloads	N/A	Static web server
jbb-1	Wisconsin Commercial Workloads	N/A	specJBB-like middleware transaction processing
oltp-1	Wisconsin Commercial Workloads	N/A	TPC-C-like OLTP
zeus-1	Wisconsin Commercial Workloads	N/A	Static web server

Eight-way threaded Commercial Workloads [5] and microbenchmarks (described in Chapter 6) are used in this thesis to evaluate proposals in the context of multi-threaded workloads, summarized in Table 3-4.

TABLE 3-4. Descriptions of Multi-Threaded Workloads

Benchmark	Suite	Description
`apache-8`	Wisconsin Commercial Workloads	Static web server
`jbb-8`	Wisconsin Commercial Workloads	specJBB-like middleware transaction processing
`oltp-8`	Wisconsin Commercial Workloads	TPC-C-like OLTP
`zeus-8`	Wisconsin Commercial Workloads	Static web server

Unless otherwise noted, all benchmarks are run for a minimum of 100 million instructions. Before the run, workloads are fast-forwarded past their initialization phases. Caches, TLBs, and predictors are warmed before profiling begins. Multiple runs are used to achieve tight 95% confidence intervals for multi-threaded runs; single-threaded runs show little variability—differences in initial conditions affect only branch predictor state and is almost entirely homogenized by the warmup process, and random latencies added to memory latencies do not affect average latency. In other words, divergence between runs is rare in single-threaded workloads, and accounted for via multiple runs in multi-threaded workloads [6].

Chapter 4

Serialized Successor Representation

This chapter discusses Serialized Successor Representation (SSR), a graph-based, name-free dataflow dependence representation for register dependences. In this chapter, SSR is treated in isolation of any hardware artifacts present in a particular core implementation—instead, the evaluation in this chapter focuses on the principal tradeoffs of SSR: a larger scheduler at the cost of serialized wakeup of multi-successor values.

Initially, SSR is described in the abstract case of an unbounded computation (Section 4.2). I then present arguments detailing how SSR can be used to guide arbitrary-length out-of-order executions using a finite instruction window (Section 4.2.1), and how to use SSR to represent dependences between memory operations (4.2.2).

I also consider the performance implications of SSR on cores that use this dependence representation, idealizing hardware as necessary to demonstrate the potential performance advantages of the schedulers made possible by SSR (Section 4.3).

4.1 Naming in Out-of-Order Microarchitectures

Named architectural state has been a part of computer architecture at least since the earliest so-called von Neumann machines, e.g., Harvard Mark I [2], the EDVAC [177], and others. Names have taken the forms of register identifiers—explicitly in enumerated register files (e.g., `eax` in x86, `%l0` in SPARC, `$R9` in MIPS, etc.), as well as the form of implicit registers (e.g., accumula-

tors)—and as addresses in architected memories (each addressable location has a name, and occasionally, more than one name). Initially, these names referred to individual locations in the machine's storage systems, but as computers become more sophisticated and ISAs remained mostly unchanged, various optimizations changed the meaning of these names—in particular, the introduction of physical register files and cache hierarchies broke the abstraction that a name corresponds to unique location in a given design. Instead, the *state* retains the name, but its *location* may change over time. For instance, register renaming[1] arose in order to resolve WAW dependences on individual architectural register names, by associating different dynamic values with different physical register names.

Named state has indirectly lead to poor scalability of several key components of modern microarchitectures, largely out of the need to reconcile an architectural name with its (current) physical location. In particular,

- The **Physical Register File** (PRF) implements the namespace of physical registers in a modern CPU. Physical register names are associated in the microarchitecture with dynamic values. The PRF is typically on the critical path of instruction issue; it is read after instruction selection, and before bypassing. All functional pipes must access a global register space at issue-time, to ensure the correct input operand values are read. This is troublesome in a large core design, because the PRF must service reads and writes from several functional units, which may not be near one another on the die. Distributing the PRF, e.g., in a clustered design [126, 30, 20], gives rise to significant complexity.

1. Arguably, Tjaden and Flynn were the first to introduce the concept of renaming [166], though the name was coined by R. M. Keller five years later [93]. Dezsõ Sima gives an overview of renaming techniques [150].

- **ALU bypassing** relies on broadcasting a physical register identifier, along with an associated value, to all other ALU input ports in the processor core that could potentially consume the value. It is well known that this approach is not scalable, as the complexity of routing and register tag comparison quickly grows prohibitive as the number of endpoints increases (e.g., the number of comparisons and the number of wires grows quadratically in the number of endpoints in the bypassing network [68]).
- **Instruction scheduling** commonly relies on CAM-based [190] or matrix-based [70, 144] associative search, limiting the scalability of the traditional instruction scheduler. Physical register identifiers are associated with successor instruction operands; a search for dependent operands is performed on the scheduler whenever physical registers are written.
- **Memory disambiguation** is typically implemented with a combination of dependence prediction [119, 38, 159] and associative search of a load/store queue [56, 127, 158]. The latter constitutes a search of memory target addresses belonging to all other in-flight memory operations. Similar to the previous microarchitectural components, which operated on physical register identifiers, the disambiguation subsystem must identify memory locations with identical (or overlapping) addresses (i.e., memory names).

Some of these shortcomings have been the focus of prior work in the specific areas mentioned. For instance, prior work in instruction schedulers [78, 136] using a single first-successor pointer make the key insight that explicit successor representation can *reduce the frequency* of broadcast operations, limiting the associated dynamic power cost as well. NoSQ [146], which relies entirely on prediction for disambiguation, ameliorates the need for a costly centralized LSQ altogether.

However, NoSQ still relies on naming register values with physical register IDs, which again gives rise to associative search in schedulers, and these register values reside in a centralized PRF.

In all of the above structures, values are associated with name—either a physical register identifier, or a memory address—and this "name" accompanies the value throughout the pipeline. This *association* tends to give rise to *broadcast* and centralization, which in turn limit scalability of the underlying microarchitecture. It is the need to eliminate broadcast operations and centralization that motivates a new abstraction for representing inter-instruction data dependences.

4.2 Serialized Successor Representation (SSR)

Fundamentally, the notion of a broadcast is unnecessary. In practice, it is also wasteful, as dynamic values tend to have few successor operations [136, 144]. Though some prior schedulers have leveraged this observation by implementing pointer-based schedulers [132, 136, 176] to optimize the common case, this work is used pointers to handle *all* cases.

Serialized Successor Representation (SSR) is a method of representing inter-instruction data dependences without any association. Instead of maintaining value names, SSR describes values' relationships to operands of other instructions. In SSR, distributed *chains of pointers* designate successors of a particular value. Instructions in SSR are represented as *three-operand tuples* (i.e., instructions are in three address form): SOURCE1, SOURCE2, and DESTINATION, abbreviated as S1, S2, and D. Each *operand* consists of a *value* and a *successor pointer*. This pointer can designate any other operand, belonging to some subsequent instruction (either S1, S2, or D), or can be NULL.

Operand pointers are used to represent data dependences. In particular, register dependences are represented as pointers from the producer operand to the first node of a linked list of successor operands. The pointer field of the producing instruction's D-operand designates the first successor operand, belonging to a subsequent successor—usually the S1- or S2-operand of a later instruction. If a second successor exists, the pointer field at the first successor operand will be used to designate the location of the second successor operand. Similarly, the locations of subsequent operands can be encoded in a linked-list fashion, relying on the pointer at successor *i* to designate the location of successor *i+1*. The last successor is marked by a NULL pointer field. Figure 4-1 illustrates SSR over a simple sequence of pseudo-assembly code (note, destination registers are on the right-hand side of the figure).

A topological sort of all instructions in a SSR graph yields an out-of-order execution that respects data dependences between registers. The executions yielded by topological sorts in SSR are a subset of those arising from true dataflow, as SSR will impose program order on instructions

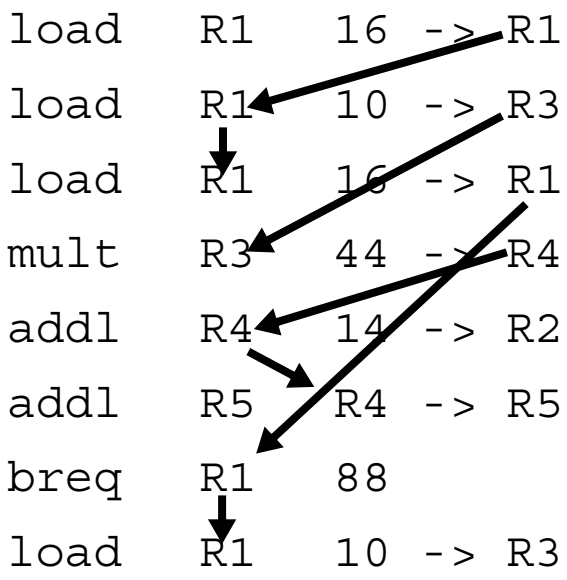


FIGURE 4-1. Simple pseudo-assembly sequence with SSR pointers superimposed.

that consume the same value, whereas a true dataflow graph will not. Importantly, program order is always a correct topological sort of an SSR graph.

SSR is similar to dataflow representations like those used in the MIT Tagged Token Machine [14] and in the EDGE ISA [142, 94], in that it is a pointer-based dataflow representation. However, unlike proposals requiring software, ISA, or compiler changes, SSR is intended to be derived dynamically in hardware from a conventional von Neumann (mainstream) ISA, using a process similar to contemporary register renamers. However, the advantage of SSR is not in renaming, but rather in *de-naming*.

SSR's pointer-based representation of dependences never requires a search or broadcast operation to locate a successor for any dynamic value—successor operands are always *explicitly* and unambiguously identified. Dynamic values in SSR no longer have a single name (e.g., architectural register *R4* or physical register *P9*)—instead, multiple instances of the same static value reside at each producer or successor operand. The implication of explicit representation to hardware is that an SSR-based scheduler is not associative: it can be built from simple SRAMs, thereby reducing complexity, saving power, and enabling the scheduler to scale to large sizes.

4.2.1 SSR in Finite Hardware

SSR applied to an entire computation produces a DAG, which can be topologically sorted to yield an execution respecting in-program-order register semantics. However, it is not possible to implement hardware capable of capturing a complete SSR representation for an arbitrary computation. In particular, a modern computer system can be viewed as a particular instance of the halting problem [171]—the entire computation of a modern system might, in theory, never terminate. Capturing the entire SSR representation would require unbounded hardware.

Therefore, in order to realize SSR as a practical means to represent dependences, it must be argued that SSR-based cores can generate a correct execution even if the entire computation cannot be simultaneously represented in bounded hardware. The argument addressing finite hardware is inductive in nature, and is based on a two simple premises: First, the class of computations representable by SSR are those that produce transformations on a state space consisting of discrete registers, nominally represented as a set of register name/value pairs. This discussion will assume these registers are organized into a register file (RF), and therefore SSR computations map register file states to other register file states through a set of transformations, e.g., $RF_A \rightarrow RF_B$. Second, these transformations obey transitivity.

Consider the architectural register state of a machine, after executing i instructions. Call this state ARF_i (i.e., "architectural register file after i instructions"). An unbounded number of instructions may follow the i^{th} instruction—call these following instructions $[i, \infty)$, as instruction numbering starts at zero. Suppose an SSR DAG is constructed for the W instructions following the i^{th} instruction—such a DAG represents the dependences on the interval $[i, i + W)$. If this DAG was used to guide an execution, the resulting transformation on the register space would be $ARF_i \rightarrow ARF_{i+W}$. Once an execution has determined the definition of ARF_{i+W}, the process could be repeated for interval $[i + W, i + 2 \cdot W)$, and so on for an arbitrary number of subsequent instruction intervals.

The above argument still holds even if the actual instructions W constitute a *sliding* window over $[0, \infty)$. Consider again an SSR DAG over W instructions $[i, i + W)$, representing transformation $ARF_i \rightarrow ARF_{i+W}$. The complete transformation $ARF_i \rightarrow ARF_{i+W+1}$ can be represented as SSR computations over $[i, i + 1)$ and $[i + 1, i + W + 1)$, yielding $ARF_i \rightarrow ARF_{i+1} \rightarrow ARF_{i+W+1}$, by

transitivity. In other words, removing one instruction i from $[i, i+W)$ and appending instruction $i+W$ yields another valid SSR DAG of size W over instructions $[i+1, i+W+1)$, representing the transformation $ARF_{i+1} \rightarrow ARF_{i+W+1}$. Again, this process can continue arbitrarily, never requiring simultaneous representation of more than W instructions at a time, by leveraging finite register state.

The implication of this argument is that it is possible to construct an SSR-based instruction window in bounded hardware, because only a subset of dependencies must be retained if a correct record of prior register transformations (i.e., an architectural register file) is maintained. All subsequent instructions must represent transformations over this state.

4.2.2 SSR for Memory Dependences

Thus far, this discussion has considered only the representation of register dependences with SSR—store-to-load dependences through memory have received no treatment. However, processor microarchitectures maintaining more than a handful of in-flight instructions must provide a mechanism to identify load operations that address the same memory as prior in-flight store operations, and must ensure those loads observe the correct value in predicted program order. This is the conventional role of the memory disambiguation subsystem, commonly implemented with a CAM-based load/store queue, augmented with dependence prediction [119, 38, 159, 56, 127, 158].

At least two proposals in the literature present a complete alternative to associative search: NoSQ [146] (literally No Store Queue) and Fire-and-Forget [160]. This work adopts a solution based on the former. Sha et al. propose NoSQ as an implementation of uniprocessor memory disambiguation for a conventional core microarchitecture, relying on register renaming. The premise behind NoSQ is that store-to-load forwarding is easily predictable, based on past behavior. After

training, NoSQ's predictor identifies store/load pairs likely to forward, and dynamically adjusts the dataflow dependences of the load such that, instead of accessing memory, the load operation reads the physical register specified by the store as its data operand. Because NoSQ is a form of speculative memory bypassing, its predictions require verification prior to commit. In most cases, predictions can be verified by comparing *store sequence numbers* (SSNs) between prior (committed) stores and committing loads, which retain the SSN of the store from which they forward data. Corner cases in NoSQ are handled by commit-time value replay [29], in which vulnerable loads re-read their values from the L1-D cache at commit-time. Loads improperly disambiguated observe a value different than that present in their destination register. Resolution of incorrect disambiguation is implemented via a pipeline flush.

Two obstacles present themselves when adapting the NoSQ approach to an SSR-based core. First, aside from control prediction, SSR is a non-speculative representation of inter-instruction dependences. However, given that NoSQ includes a mechanism to verify predictions (load vulnerability filtering, and load replay when necessary), as well as a mechanism to handle mispredictions (full pipeline squash), false predictions can be guaranteed never to affect architectural state (though the SSR graph must be rebuilt or otherwise purged of incorrect dependences). Second, SSR-based cores have no register renamer—nor even a register namespace in the SSR-equipped portion of the core—hence, another means is required to establish store-to-load dependence.

The latter is accomplished by leveraging the fact that operations in SSR must be represented in three-address form. Though contemporary ISAs include mechanisms to write memory that do not conform to three-address form [82, 181], these store operations can be represented as a separate address calculation and store to memory, which *are* each three-address-form operations. The key

exploitable property of a three-address form store operation is that it has no destination operand, as the store does not cause a register write.

It is possible to establish a (speculative) dependence between a store and a subsequent prediction load operation in SSR by establishing a pointer from the store's destination operand (guaranteed by construction to be unused) to the destination operand of the (predicted) dependent load operation. The load itself should be marked as cancelled, such that it does not actually execute when its source operands are made available—instead the load appears to execute as soon as the forwarding store's address and data are known. Establishing a possibly false dependence between a store's destination operand and a subsequent load's destination operand manifests the predicted data dependence between the operations, and yet still preserves the data required to replay the load, if necessary, at commit-time in order to validate the prediction (i.e., the load maintains its original input operands, hence the load address can still be calculated). Figure 4-2 illustrates this process with a simple example, in which the store-data operation (`stwd`) forwards to a subsequent load (`load`) into `R3`. It would be possible to use the data operand of the store operation instead of

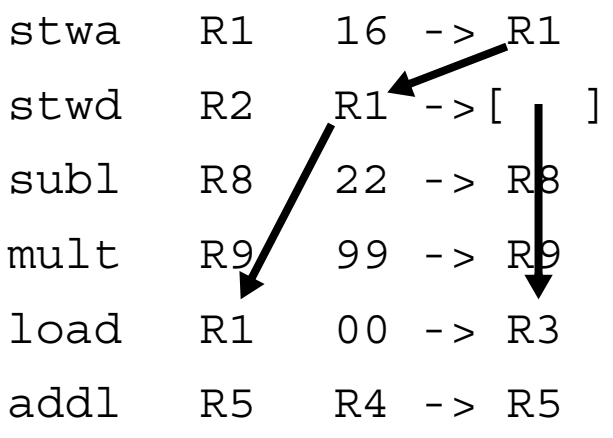


FIGURE 4-2. Store-to-Load Forwarding represented in SSR.

the destination operand for the same purpose, provided non-speculative successors are handled, but leveraging the store's data operand register is more intuitive.

Cores implementing NoSQ inherit the disadvantages of the original NoSQ proposal. In particular, loads that forward from multiple prior stores cannot be represented within the framework of a single register dependence (though stores that forward to several subsequent loads in predicted program order *can* be represented by a linked list of successor operations). Furthermore, stores that only partially overlap later loads cannot be adequately represented. All unhandled cases manifest as pipeline squashes, as these cases are naturally detected by vulnerability filtering [139] and replay.

4.3 Performance Implications of SSR

SSR serializes instructions that consume the same dynamic value. Prior work has shown that values with multiple successors are uncommon [136, 144]. I have performed a similar analysis on the workloads used in this thesis, and arrived at a similar conclusion: most dynamic values (approximately 80%) have one or zero nearby successors (e.g., within 500 committed dynamic instructions). Despite the relative rarity of multi-successor values, there is nonetheless a performance loss incurred due to serialized wakeup, to a degree that varies by workload. For instance, integer benchmark `bzip2` includes an inner loop with a high-fanout value, which degrades performance of SSR scheduling (shown in Section 4.3.1).

However, SSR-based cores can leverage the scalability of explicit successor representations—in particular, because SSR-based cores are free of broadcast, there are fewer inherent constraints on the size of the core's scheduler than in a CAM-limited microarchitecture. Large instruction schedulers are important because they expose lookahead—the ability to examine an instruction stream

and identify independent instructions for concurrent, out-of-order execution, thereby improving core performance.

4.3.1 Quantitative Evaluation

In order to fully understand the inherent overheads of SSR, I augment of two conventional architectures with SSR—a conventional out-of-order core, similar to the Alpha 21364 [88, 41] and an RUU-based [154, 153] design (nominally *OoO* and *RUU*, respectively, from Chapter 3). Changes to implement SSR are idealized, such that only the scheduling window is affected. In order to minimize the microarchitectural changes (to isolate, to the extent possible, the effect of SSR from other side effects from microarchitectural changes), no other areas of the design are changed.

An SSR-based scheduler has two effects on the baseline core designs:

- The principal motivator behind SSR is to enable a scalable representation of scheduling information. SSR-equipped cores' schedulers span the entire instruction window, enabling improved performance through lookahead. The value of increased lookahead will vary by workload, depending on how much instruction-level parallelism is available.
- SSR serializes wakeup of successors for a given value. Serialized wakeup reduces performance, to a degree that varies by workload—codes exhibiting larger degrees of value re-use will experience greater performance degradation from serialized wakeup than codes with little re-use.

TABLE 4-1. Summary of Configurations for SSR Evaluation.

Configuration	Description
RUU	Idealized RUU-based [154] microarchitecture. Renamed, fully-provisioned physical register file, 4-wide, no scheduling delays, full-window broadcast-based scheduler. Loose performance upper-bound for a given window size.
OoO	Alpha-style out-of-order microarchitecture. Renamed fully-provisioned physical file, 4-wide, broadcast-based unified scheduler size is one quarter the size of the ROB (ROB and scheduler size varied by experiment).
RUU+SSR	*RUU*, using an SSR-based scheduling representation instead of broadcast. *RUU+SSR* suffers sequential wakeup hazards as a consequence of SSR-based dependence representation, but has no resource constraints.
OoO+SSR	*OoO*, replacing the broadcast-based scheduler with a larger, full-window SSR scheduler. *OoO+SSR* has greater lookahead than *OoO*, but must wake multiple successors one per cycle. Subject to the same resource constraints as *OoO*: finite bandwidth, realistic forwarding, etc.

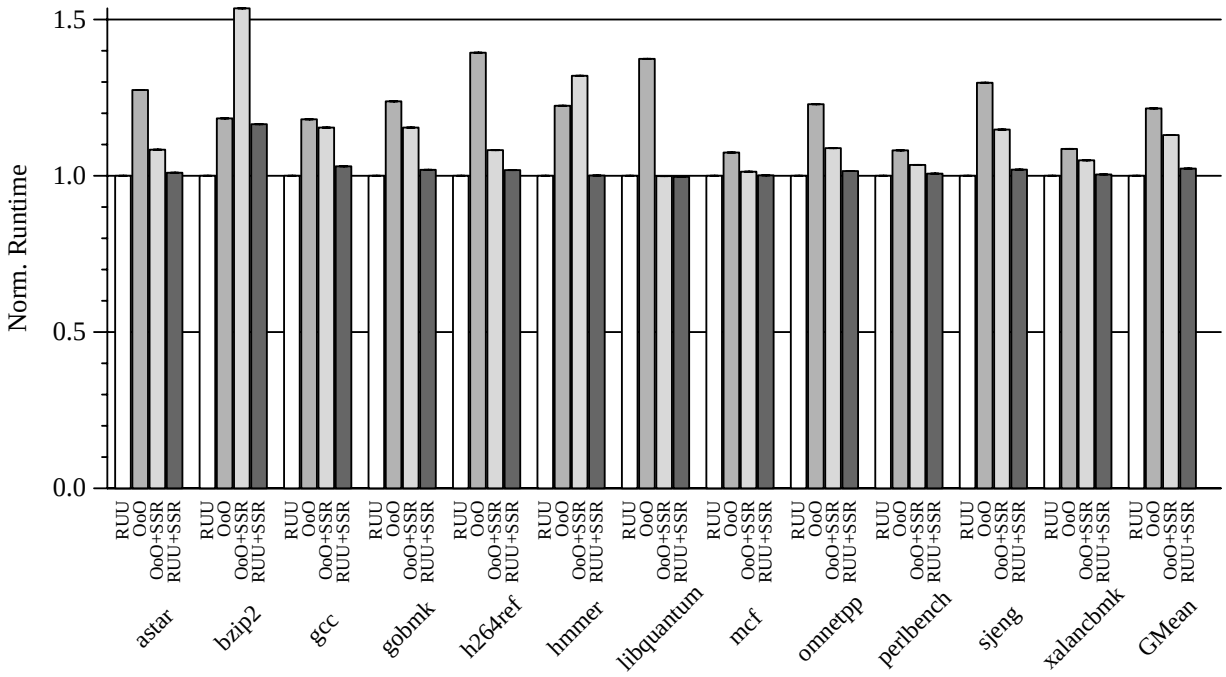


FIGURE 4-3. Normalized runtime, SPEC INT 2006, 128-entry instruction windows.

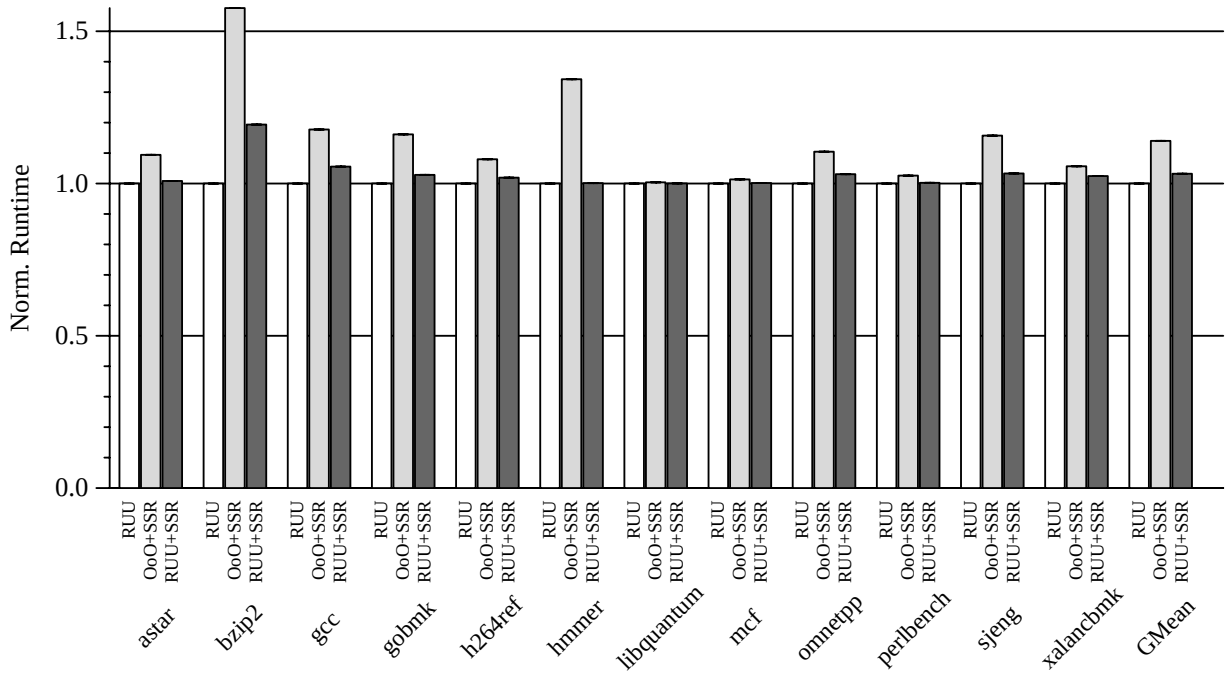


FIGURE 4-4. Normalized runtime, SPEC INT 2006, 256-entry instruction windows.

SSR-based scheduling is practical if the performance gained by the increase in scheduling lookahead exceeds the performance lost due to serialized wakeup. This tradeoff motivates the baseline architectures used in this study. *OoO*, augmented with SSR (*OoO+SSR*), is able to reap the benefit of increased scheduler size, but also suffers from a minor performance degradation due to serialized wakeup. A comparison between *OoO* and *OoO+SSR* demonstrates the practical performance advantage of SSR. The performance lost due to serial wakeup is obvious from a comparison between *OoO+SSR* and *RUU—RUU's* scheduler is sized identically, but has no serial wakeup. Because *RUU* and *OoO* differ in their degrees of idealization, especially in the organization of the functional pipelines, the *RUU+SSR* configuration represents an optimistic design without most of the resource constraints that might be expected in an actual implementation (e.g., *OoO* implements realistic issue arbitration, branch resolution, etc.). Table 4-1 summarizes the configurations used in this evaluation.

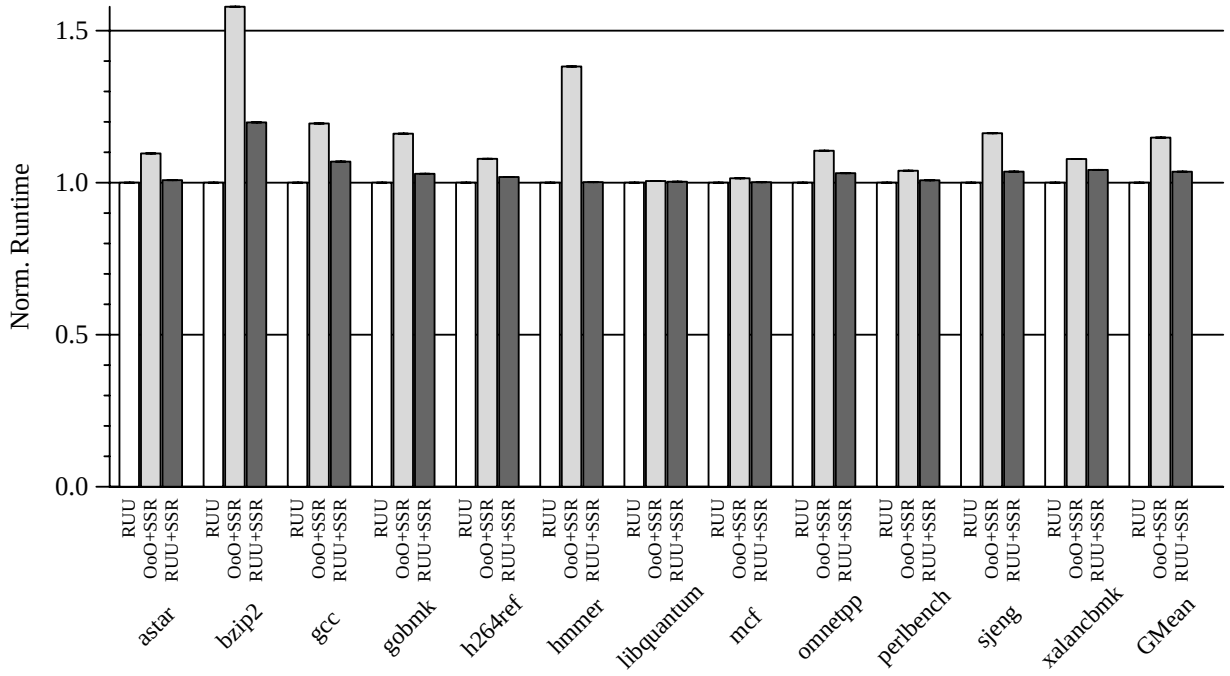


FIGURE 4-5. Normalized runtime, SPEC INT 2006, 512-entry instruction windows.

Figure 4-3 plots runtime of these four microarchitectures, normalized to that of *RUU* for SPEC INT 2006 with 128-entry instruction windows. Figures 4-4, 4-5, and 4-6, plot normalized runtime for the same experiment with 256-entry, 512-entry, and 1024-entry windows, respectively. *OoO* does not appear on plots for window sizes greater than 128, as the scheduler does not easily scale to those sizes with realistic timing assumptions. However, the key performance arguments are supported by data in the 128-entry window case (Figure 4-3). As expected, *RUU* tends to exhibit the best overall performance—it has the largest scheduling window, and suffers no wakeup latency penalties. *OoO* usually performs noticeable worse than *RUU*—though it also implements a fast wakeup, its lookahead is limited by a 32-entry scheduler. In most cases, substituting a full-window SSR-based scheduler into the *OoO* microarchitecture (i.e., *OoO+SSR*) yields a performance improvement, bringing *OoO+SSR*'s performance closer to that of the idealized *RUU*. The performance difference between *OoO+SSR* and *RUU+SSR* arises in the assumptions of the *RUU* model—

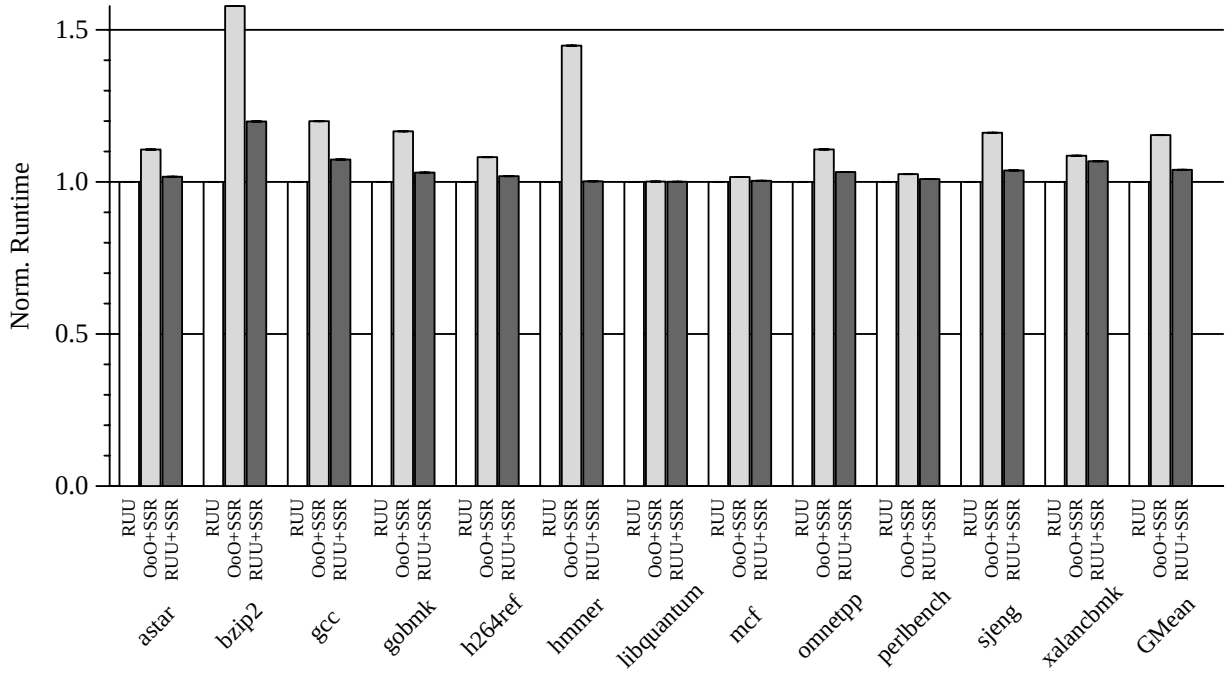


FIGURE 4-6. Normalized runtime, SPEC INT 2006, 1024-entry instruction windows.

RUU idealizes several areas of the microarchitecture, e.g., branch resolution, register file access, etc.—and from fundamental differences in microarchitectural organization (e.g., *RUU*'s ARF versus *OoO*'s PRF).

OoO+SSR degrades performance with respect to *OoO* in only two benchmarks from the SPEC INT 2006 suite: `bzip2` and `hmmer`. In the first case (`bzip2`), the data suggests the performance improvement derived from increased lookahead does not exceed the performance degradation of serialized wakeup—e.g., even *RUU+SSR* shows substantial performance degradation. In the second case, `hmmer`, is somewhat anomalous in this benchmark suite as the execution is write-bandwidth-limited, rather than window-limited. `hmmer` exhibits one of the highest IPCs of any examined workload (architectural IPC for `hmmer` on *RUU* is approximately 2.6—IPC including micro-operations is higher still). Most operations write an output register, which in turn saturates the write bandwidth of *OoO*'s physical register file.

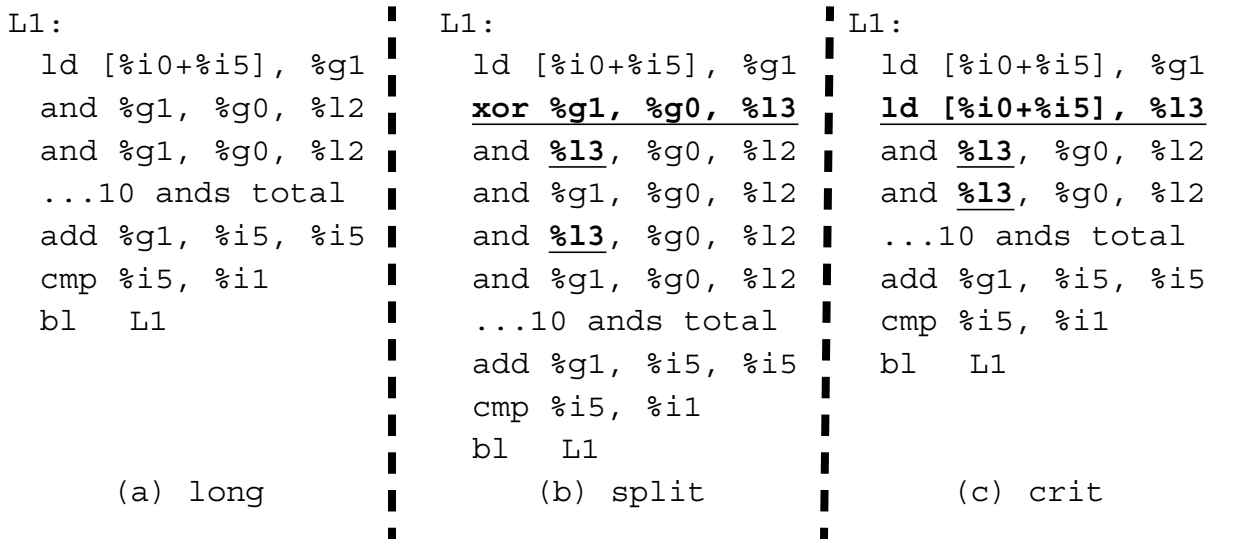


FIGURE 4-7. (a) Long-dependence chain microbenchmark `long`, (b) a simple compiler optimization `split`, breaking the chain into two chains of half length (`%l3` and `%g1`), and (c) a better optimization repeating the `ld` operation to reduce the length of the critical `%g1` chain, `crit`..

At least some of `bzip2`'s performance problems arise because of repeated use of the same architectural register, leading to a long chain of pointers and resulting in serialized wakeup. A repeated register is specifically desirable in some microarchitectures, including the UltraSPARC-III+, for which the compiler was optimizing when the SPEC INT benchmarks were compiled. In light of this, I briefly consider three microbenchmarks, evaluating possible compile-time optimizations to attack this problem in an SSR-based core. Figure 4-7 presents their assembly-level operations for these microbenchmarks. In the first case, `long`, a load operation produces register `%g1`, and ten intervening and operations consume `%g1` before `%g1` is used in the next address calculation. This leads to a long pointer chain dynamically on `%g1` in an SSR-based core.

Figure 4-7 also presents two software-level optimizations to this problem. The first, `split`, requires no software knowledge other than the detection of the long pointer chain. `split` simply reproduces another explicit instance of the same value in a different register, `%l3` (by inserting the

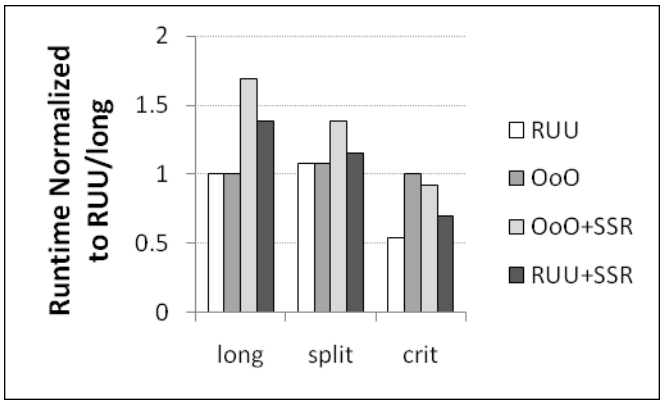


FIGURE 4-8. Runtime of `long`, `split`, and `crit`, normalized to that of *RUU* on `long`.

`xor`), and reassigns half of the successors of `%g1` to read `%l3` instead. With very little effort, this roughly halves the length of the value chain, by splitting it into two parallel paths.

More sophisticated analysis could reveal that the next address calculation, the `add`, is critical to overall performance, and choose to specifically optimize that execution path. The `crit` micro-benchmark emulates this optimization, by repeating the `ld` operation with two different output registers. One output register is used only for the critical path (i.e., the `add` successor), and the other register is used for all non-critical successors (i.e., the `and`s).

Figure 4-8 plots the runtime of each microbenchmark, normalized to that of *RUU* running `long`. *RUU* always performs best, and `crit` improves *RUU*'s performance, by increasing the likelihood that the critical path will execute first. The `split` optimization hurts the performance of both *OoO* and *RUU* slightly by inserting an extra instruction, but reduces the runtime of SSR-based cores. `crit` further reduces SSR runtime, by prioritizing the critical path with its own exclusive pointer chain.

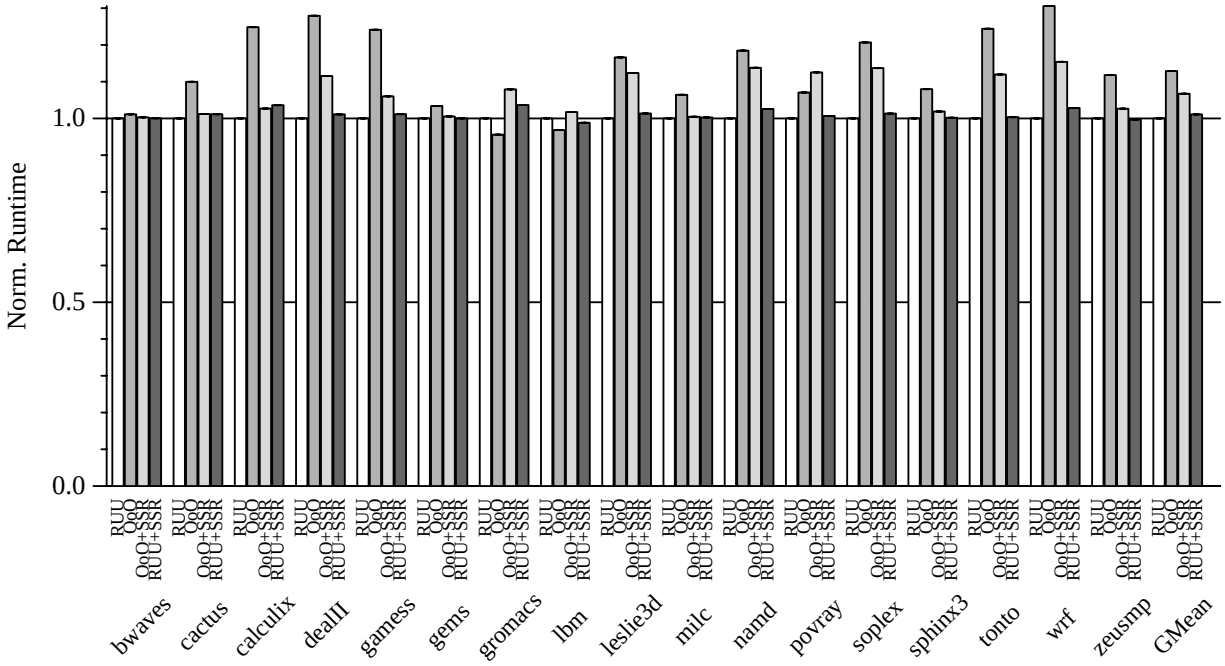


FIGURE 4-9. Normalized runtime, SPEC FP 2006, 128-entry instruction windows.

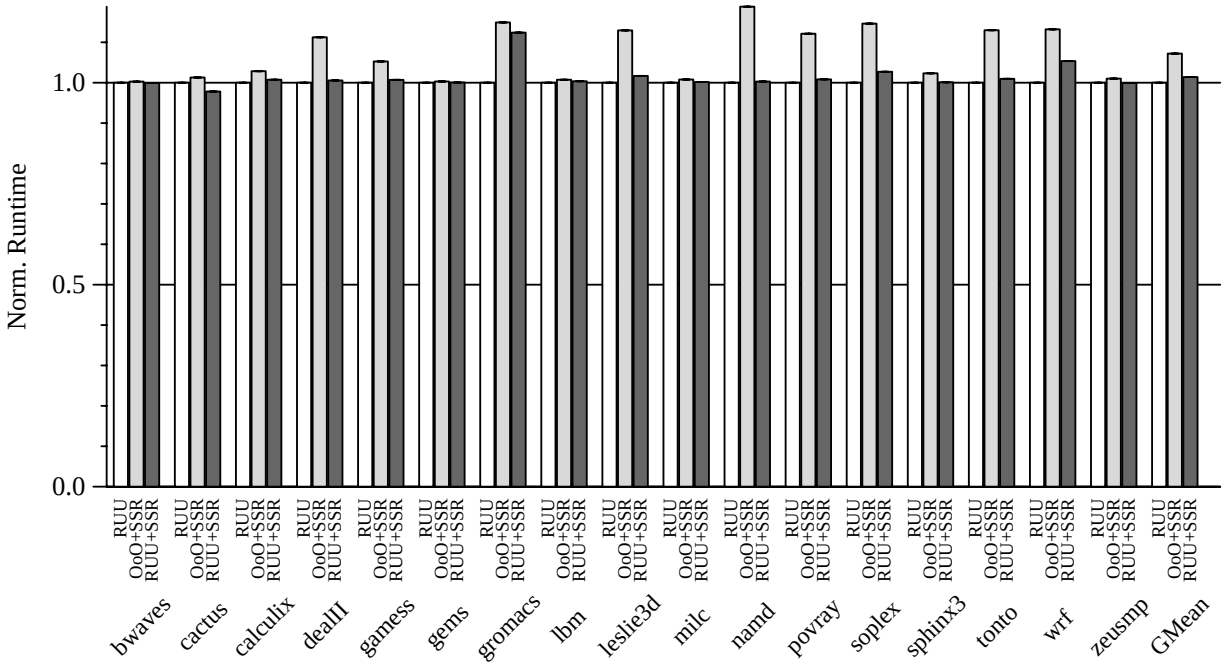


FIGURE 4-10. Normalized runtime, SPEC FP 2006, 256-entry instruction windows.

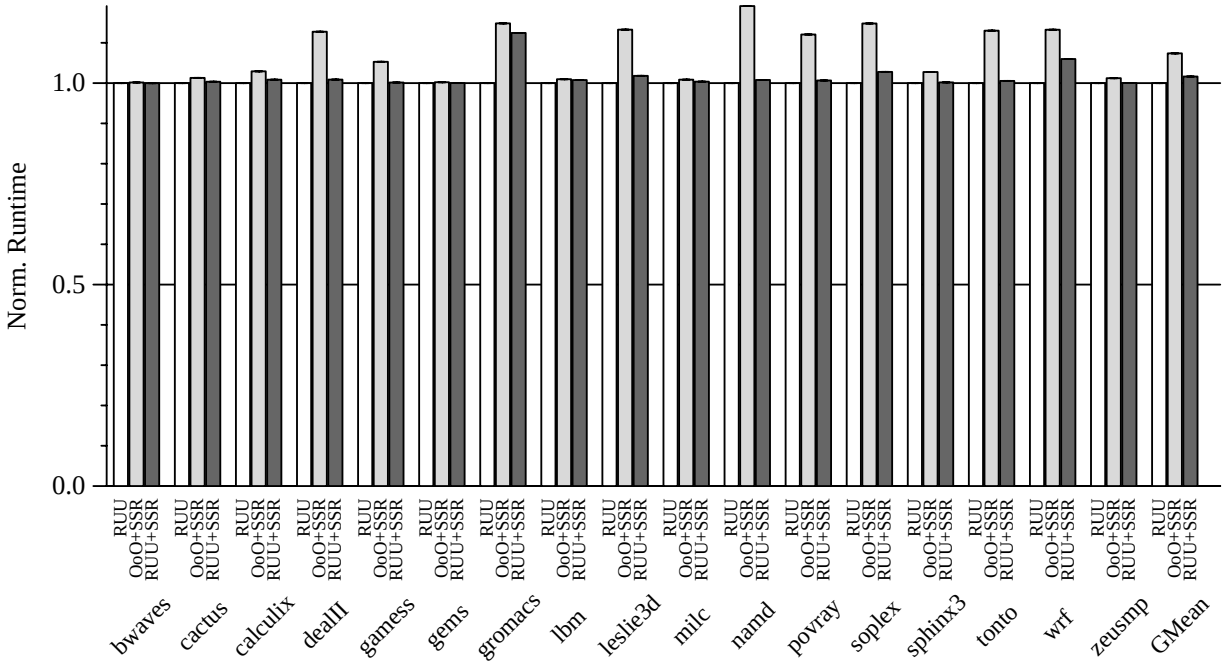


FIGURE 4-11. Normalized runtime, SPEC FP 2006, 512-entry instruction windows.

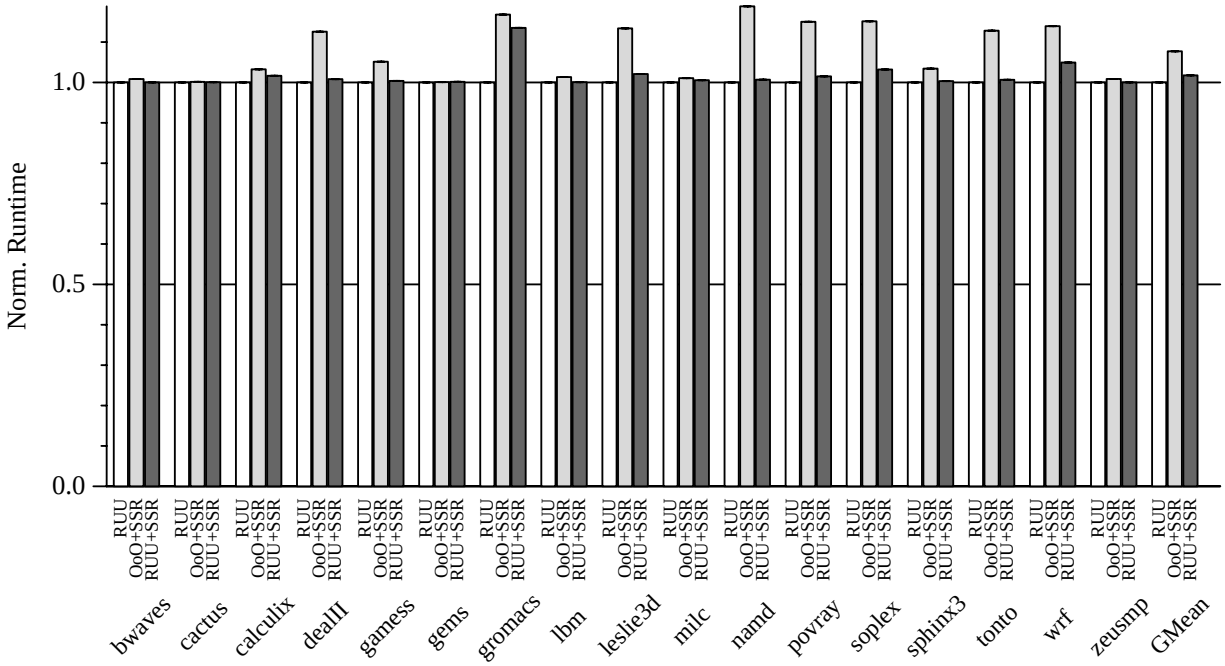


FIGURE 4-12. Normalized runtime, SPEC FP 2006, 1024-entry instruction windows.

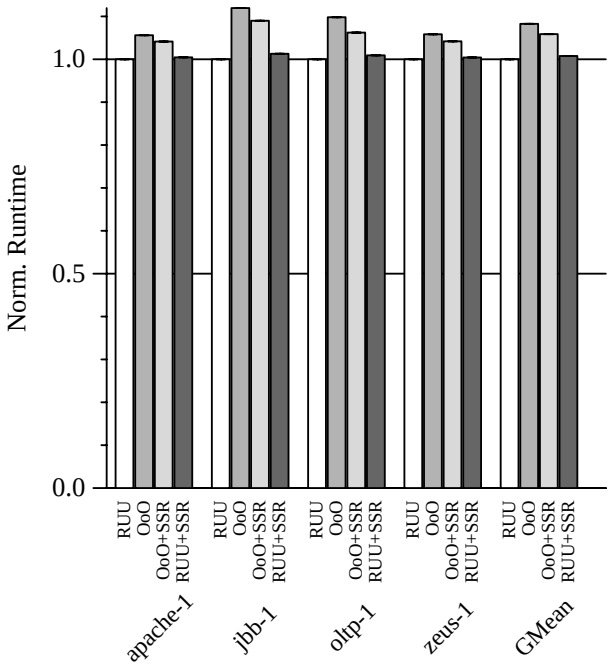


FIGURE 4-13. Normalized runtime, Commercial Workloads, 128-entry instruction windows.

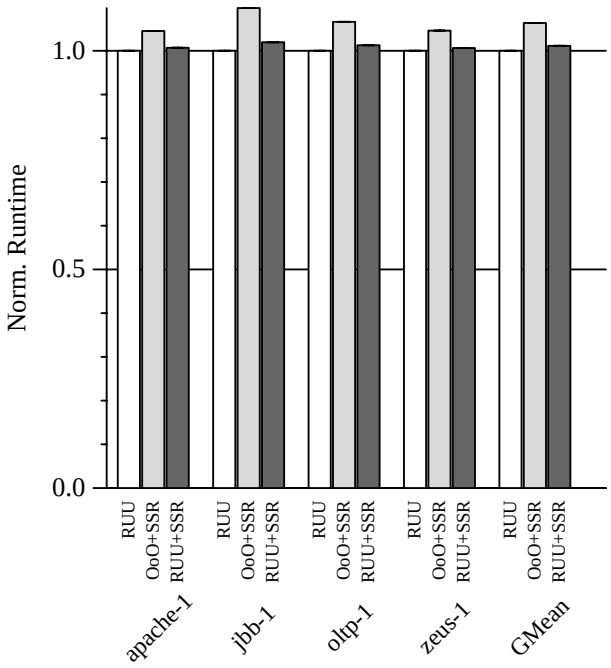


FIGURE 4-14. Normalized runtime, Commercial Workloads, 256-entry instruction windows.

The remaining graphs for SPEC INT show similar trends to those observed in the 128-entry case. However, as the window size scales, the performance degradation from serialized wakeup shrinks. This is an intuitive result: because most operations have few successors, larger windows should not necessarily imply longer dependence chains. The trend of *RUU+SSR* to remain close to the performance of *RUU* supports this claim.

Data from SPEC FP 2006 (Figures 4-9, 4-10, 4-11, and 4-12) shows similar general trends to those of SPEC INT 2006. However, two anomalous cases are present in the 128-entry window experiments: `gromacs` and `lbm` (Figure 4-9). For these two benchmarks, the *OoO* core slightly outperforms the *RUU* core. Though unusual, this small performance gap is not impossible. The *RUU* model assumes a small number of buffers are available at the head of each functional pipeline. Wakeup occurs in *RUU* when new results are available, and selection stalls only when these

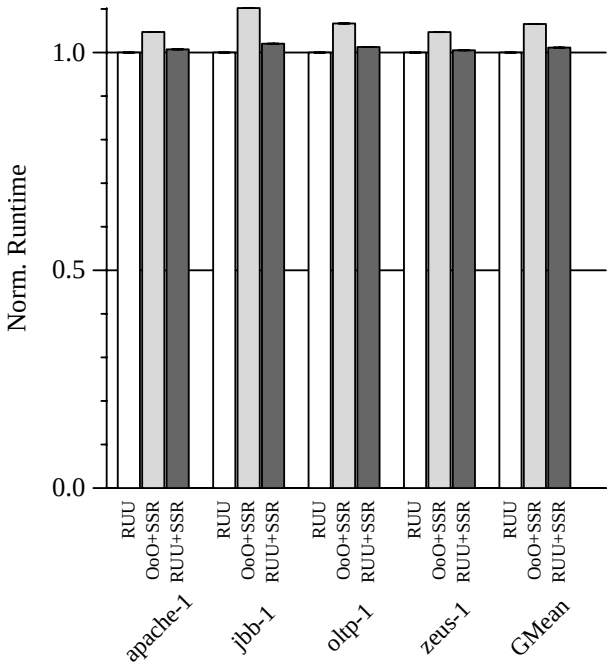


FIGURE 4-15. Normalized runtime, Commercial Workloads, 512-entry instruction windows.

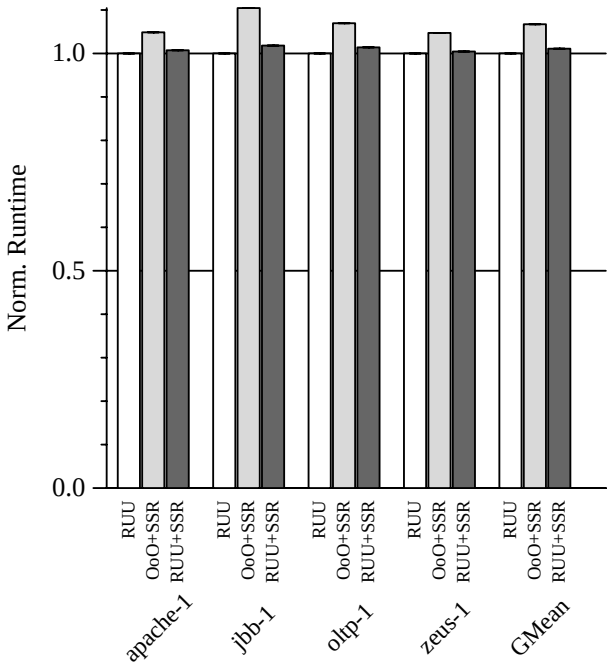


FIGURE 4-16. Normalized runtime, Commercial Workloads, 1024-entry instruction windows.

buffers are full. The implementation of pipelined wakeup in *RUU* is prescient, in that it is assumed wakeup and selection has oracular knowledge of instruction completion events (the mechanisms to queue instructions at functional units introduces variable delay on otherwise fixed-duration operations). However, *OoO* uses no such idealization. *OoO* instead implements issue arbitration concurrent to selection: Instructions do not pass selection and leave the scheduler until issue arbitration is successful (i.e., there are no queues associated with functional pipelines—variable queuing delay would significantly complicate pipelined wakeup in the *OoO* design). Under most workloads, this more-realistic set of assumptions degrades performance slightly, as scheduler space is usually a precious resource. However, for benchmarks that are not scheduler-bound, e.g., `gromacs` from SPEC FP 2006, selection inhibition can lead to better overall scheduling, by queuing instructions in the scheduler. Functional units are assumed to be pipelined and unordered (i.e.,

instructions of shorter latency can pass instructions of longer latency, if they use different physical resources). However, pipelines are subject to resource hazards, which make some schedules implicitly more efficient than others. In *OoO*, the scheduler acts as a global queue for *all* functional units, whereas *RUU* effectively employs *one* queue for each functional unit. Use of a single instruction queue allows *OoO* to occasionally exceed *RUU*'s performance due to *OoO*'s ability to dynamically generate instruction schedulers closer to optimal. However, this performance boon only appears on workloads that are not highly sensitive to scheduling space.

Lastly, for completeness, Figures 4-13 through 4-16 plot the same experiment for the Wisconsin Commercial Workload Suite. The noteworthy result from this data is that previous observations from SPEC CPU 2006 also hold for these commercial workloads.

In closing, the results above show that, for most workloads, the potential performance advantage of a large, SSR-based scheduler outweighs the performance loss incurred due to serialized wakeup of multi-successor operations, at least when considering idealized implementations of SSR-based scheduling. Chapter 5 will consider the design of a complete SSR-based core, Forwardflow, without idealization, in order to ascertain the effects of realistic resource constraints in conjunction with the performance characteristics of SSR.

Chapter 5

Forwardflow Core Microarchitecture

This chapter details the design of the Forwardflow Core Microarchitecture, and evaluates the performance of Forwardflow compared to several other core microarchitectures. This discussion provides the details of the microarchitecture underlying the scalable core implementation leveraged in Chapter 6, in a static setting, independent of the effects of scaling mechanisms and policies.

The quantitative evaluation of the Forwardflow design is fairly lengthy. I conclude this chapter briefly in Section 5.4.6, for clarity and conciseness. I then present all available data in a final section of the chapter, merely for completeness. Most readers will be satisfied with the discussion preceding the final section of this chapter.

5.1 Core Design Goals

The evolution of superscalar processing led to many studies on the available parallelism within instruction streams, arising from instruction-level parallelism (ILP) (e.g., [178, 90, 151, 166]). These studies have shown, albeit with some exceptions, that *average ILP* tends to be fairly low—"rarely exceed(ing) 3 or 4" (David Wall) [178]. Wall further observes that "...the payoff of high *peak* parallelism is low if the *average* is still small" (emphasis added). Low average ILP, despite occasional high peak ILP, argues that there is a practical limit to the useful *width* of a processing pipeline, and that architects will see diminishing returns from very wide pipelines for all but a handful

of workloads. This follows intuitively from the Forced Flow Law [96] as applied to the processor pipeline: if instructions can only retire from a pipeline at a fixed rate (the ILP limit), widening the intermediate pipeline stages (e.g., the issue width) will not change average instruction commit rate. In other words, for many workloads, peak issue rate is not a bottleneck in a superscalar core. Instead, architects should focus on improving average ILP.

One approach to this goal seeks to reduce the likelihood that a processor will stall, waiting for latency memory operations. Large-window exemplify this approach, by focussing not peak issue width but on locating ILP in the first place, by increasing lookahead. A core's lookahead determines the number of instructions the core examines concurrently to identify those suitable for parallel execution [37]. Increasing lookahead improves average ILP by looking farther into the future instruction stream to find independent work.

The pursuit of increased lookahead eventually led to designs optimized for exposing memory-level parallelism (MLP), the concurrent servicing of long-latency load misses. Proposals like Runahead Execution [46, 122] and Continual Flow Pipelines [156] exploit MLP by discarding or deferring dependent instructions in pursuit of independent work (even to the detriment of *peak* ILP, e.g., with instruction deferral or extra pipeline squashes), in order to better amortize the cost of long-latency misses.

The principal goal of the Forwardflow Core Microarchitecture is to achieve the memory lookahead of Runahead-like and CFP-like designs, without sacrificing the ILP capabilities of conventional large-window machines. To do so, the design must be capable of examining many instructions concurrently for execution (to find ILP and MLP), and must be able to issue instructions at fine granularity (i.e., instructions should wake and issue quickly, once their dependences

are resolved). However, the ability to deliver very high peak (or sustained) ILP is an explicit non-goal. Instead, this work seeks further performance gains through increased lookahead, without seeking to improve pipeline width.

Secondary to the goal of improved performance, Forwardflow cores must be more energy-efficient than traditional microarchitectures, to motivate the additional costs of aggressive cores. Power constraints now dominate chip design, and consequently, new core designs for CMPs must employ energy-efficient microarchitectural structures. A key guiding principle of this work was to avoid associative structures and wasteful broadcasts whenever possible—in part made possible by leveraging SSR—and in part made possible with disaggregated and modular instruction window design.

5.2 Forwardflow Overview

Chapter 4 detailed Serialized Successor Representation (SSR), an explicit successor representation enabling the construction of cores with both large windows and simple execution management logic. The Forwardflow Core Microarchitecture is one such implementation of an SSR-based core, which focuses on the core design goals outlined above (Section 5.1). Notably, Forwardflow's microarchitecture emphasizes the principles of locality, by avoiding centralized structures in the execution management logic. Forwardflow requires no centralized register file, no LSQ, and no global bypassing networks.

As an SSR-based core, dependences in Forwardflow are represented as linked lists of *forward pointers* [132, 136, 176], instead of using physical register identifiers to label values. These pointers, along with values for each operand, are stored in the *Dataflow Queue* (DQ), shown in Figure 5-1, which takes the place of the traditional scheduler and centralized physical register file. Instead

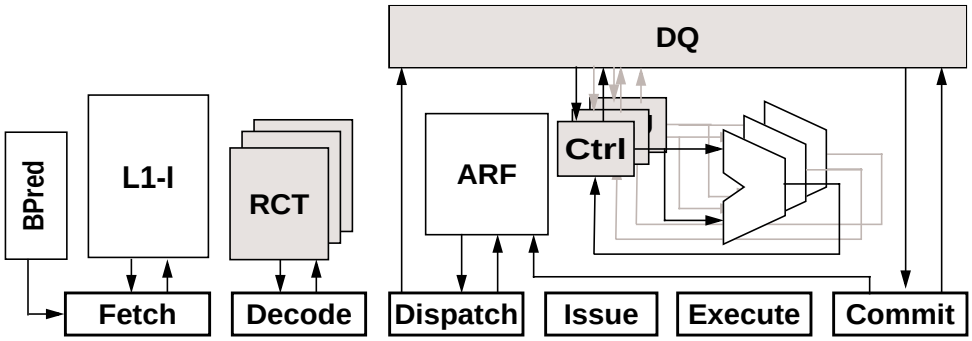


FIGURE 5-1. Pipeline diagram of the Forwardflow architecture. Forwardflow-specific structures are shaded.

of broadcasting physical register tags, DQ update hardware chases pointers to implement instruction wakeup (an example is provided in Figure 5-7 on page 97). Though most dependence lists are short, serializing wakeup causes some slowdown. However, the use of pointers throughout the design enables a large, multi-banked DQ implementation, in which independent lists are chased concurrently.

At the highest level, the Forwardflow pipeline (Figure 5-1) is not unlike traditional out-of-order microarchitectures. The Fetch stage fetches instructions on a predicted execution path, and Decode detects and handles potential data dependences, analogous to traditional renaming. Dispatch inserts instructions into the Dataflow Queue (DQ) and instructions issue when their operands become available. When instructions complete, scheduling logic wakes and selects dependent instructions for execution. Instructions commit in-order from the DQ.

Since the DQ is built entirely of small SRAMs, it can scale to much larger aggregate sizes than a traditional instruction scheduler, yet is accessed at finer granularity than a ROB. Each entry in the DQ requires an estimated $200 + 3 \cdot \lceil \log_2 N_{DQEntries} \rceil$ bits of storage. This encompasses up to three 64-bit data values, pointers to represent dependences, and control bits.

5.3 Forwardflow Detailed Operation

In this section, I discuss the operation of each stage of the Forwardflow pipeline in detail. The Forwardflow microarchitecture differs from a canonical out-of-order microarchitecture (e.g., Section 2.2.1) in many ways. Though the discussion herein generally refers to pipeline stages by their function, e.g., "Decode" or "Execute", these operations themselves are, in general, pipelined. The discussion here discusses the operations performed in each stage, without detailing how these operations are decomposed into individual clock cycles.

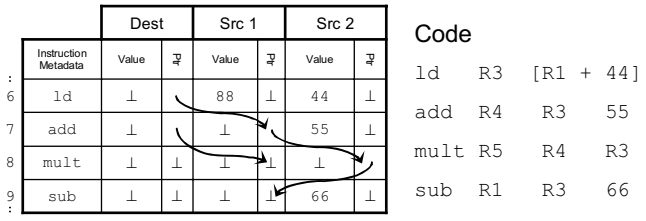


FIGURE 5-2. Dataflow Queue Example.

The *Dataflow Queue (DQ),* is the heart of the Forwardflow architecture, and is illustrated in Figure 5-2. It is involved in instruction dispatch, issue, completion, and commit. The DQ is essentially a CAM-free Register Update Unit [154], in that it schedules and orders instructions, but also maintains operand values. Each entry in the DQ holds an instruction's metadata (e.g., opcode, ALU control signals, destination architectural register name), three data values, and three forward pointers, representing up to two source operands and one destination operand per instruction. Value and pointer fields have empty/full and valid bits, respectively, to indicate whether they contain valid information. Dispatching an instruction allocates a DQ entry, but updates the pointer fields of *previously* dispatched instructions. Specifically, an instruction's DQ insertion will update zero, one, or two pointers belonging to *earlier* instructions in the DQ to establish correct forward dependences.

Much of this discussion focuses on the detailed operation of the DQ, as it is the most prominent component in the Forwardflow design. Importantly, the DQ is disaggregated—decomposed

into multiple, banked SRAMs, coupled with nearby functional datapaths, and organized into discrete units, called *bank groups*. Bank groups each implement a contiguous subsection of DQ space. *Pipeline width* is determined by the throughput of an individual bank group. *Aggregate window size* is a function of the number of bank groups.

Disaggregated design optimizes local operations, and bounds access time of the SRAMs constituting the DQ as a whole. Bank groups operate collectively during dispatch, wakeup, and commit operations. Each bank group corresponds to a different adjacent subset of DQ space (i.e., high-order bits of the DQ entry number determine the corresponding bank group). Within a group, DQ space is divided in an interleaved fashion among individual DQ banks. The details underlying this design is more fully discussed in Section 5.3.7.

5.3.1 Decode to SSR

In a Forwardflow core, the fetch pipeline (i.e., Fetch in Figure 5-1) operates no differently than that of other high-performance microarchitectures. Decode produces all information needed for Dispatch, which inserts the instruction into the DQ and updates the forward pointer chains. The decode process must determine to which pointer chains, if any, each instruction belongs. It does this using the Register Consumer Table (RCT), which tracks the *tails* of all active pointer chains in the DQ. Indexed by the architectural register name, the RCT resembles a traditional rename table except that it records the most-recent instruction (and operand slot) to *reference* a given architectural register. Each instruction that writes a register begins a new value chain, but instructions that read registers also update the RCT to maintain the forward pointer chain for subsequent successors. The RCT also identifies registers last written by a committed instruction, and thus which values can be read at the conclusion of the decode pipeline from the Architectural Register File

(ARF). Values read from the ARF are written into the DQ's value arrays during dispatch. A complete example of the RCT's operation, as well as that of the dispatch logic, is provided in Section 5.3.2.

The RCT is implemented as an SRAM-based table. Since the port requirements of the RCT are significant, it must be implemented aggressively and with some internal duplication of its data arrays. Fortunately, the RCT itself is small: each entry requires only $2 \cdot \lceil \log_2 N_{DQEntries} \rceil + 4$ bits. Furthermore, the RCT is checkpointed in its entirety on each branch prediction, for later possible

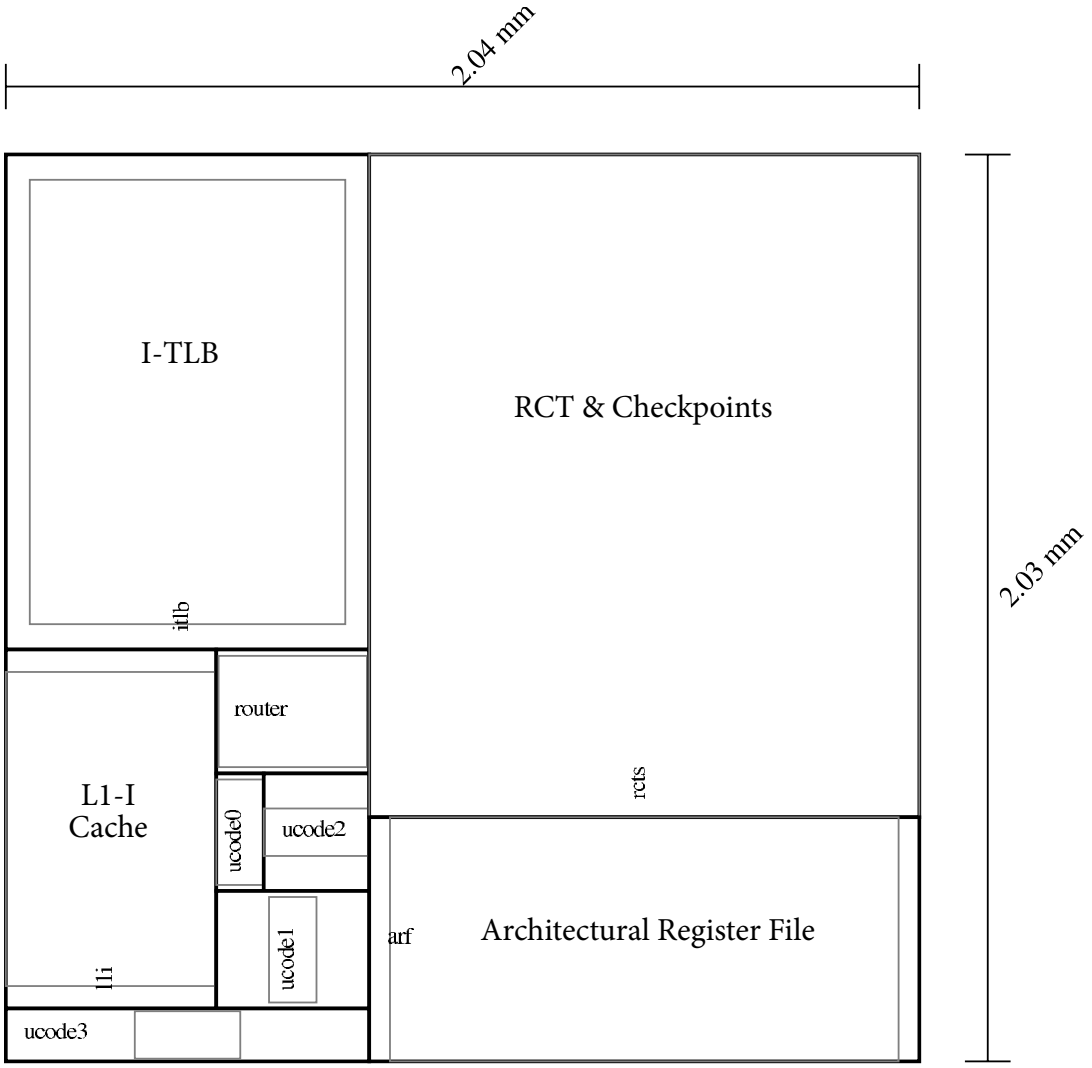


FIGURE 5-3. Floorplan of a four-wide Forwardflow frontend in *32nm*.

recovery (details in Section 5.3.5). These checkpoints account for most of the area of a Forwardflow frontend, as evident from the plot in Figure 5-3. Together with the L1-I cache, RCTs, the ARF, and decode logic, the total area of a Forwardflow frontend is approximately 4 mm^2 in *32nm* technology.

5.3.2 Dispatch

The dispatch process in Forwardflow differs from a conventional microarchitecture in several ways. Notably, large DQ designs are physically disaggregated—the dispatch logic must be aware of the physical characteristics of the DQ and operate accordingly. Figure 5-5 (on page 94) exemplifies a disaggregated DQ design, consisting of two bank groups, each in turn consisting of four DQ banks and associated control circuitry. For simplicity, I begin this discussion with a high-level description of the logical dispatch operation, followed by the details needed to implement dispatch in a disaggregated DQ.

Logical Behavior. To implement dispatch, Forwardflow's dispatch logic performs two basic operations:

- *DQ Entry Allocation:* Dispatching instructions write instruction metadata and available operand values into their assigned DQ entry. During this process, valid bits on pointer arrays are cleared, empty/full bits on value arrays are set or cleared as appropriate. The DQ tail pointer is updated to reflect the number of instructions that dispatch in a given cycle.
- *Dependence Annotation*: Dispatching instructions with register dependences require writes to the DQ's pointer arrays. Specifically, the contents of the RCT identify immediate predecessor instructions for each source register operand belonging to each dispatching instruction (i.e.,

DQ pointer fields of up to two *prior* instructions in the predicted instruction stream).

Figure 5-4 exemplifies the logical behavior of the dispatch process for a simple code sequence, highlighting both the common case of a single successor (the R4 chain) and the uncommon case of multiple successors (the R3 chain). Fields read are bordered with thick lines; fields written are shaded. The bottom symbol (⊥) is used to indicate NULL pointers (i.e., cleared pointer valid bits) and cleared empty/full bits.

In the example, Decode determines that the ld instruction is ready to issue at Dispatch because both source operands are available (R1's value, 88, is available in the ARF, since its busy bit in the RCT is zero, and the immediate operand, 44, is extracted from the instruction). Decode updates the RCT to indicate that ld produces R3 (but does not add the ld to R1's value chain, as R1 remains available in the

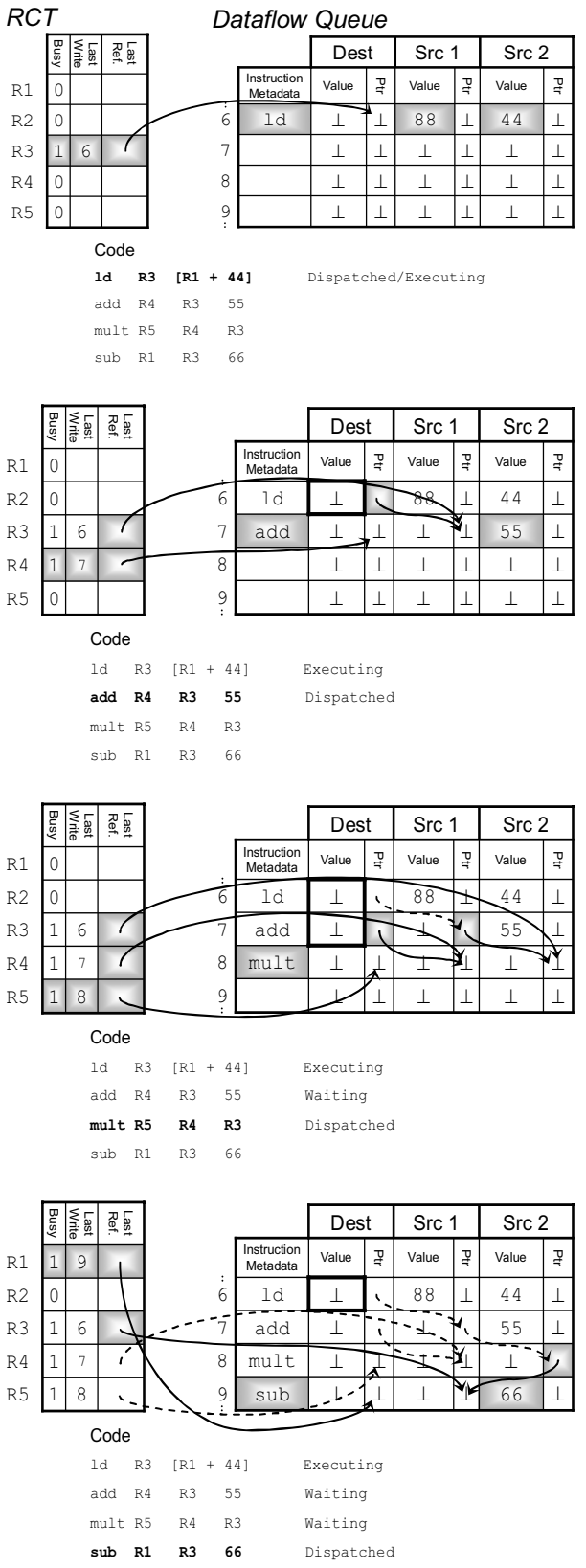


FIGURE 5-4. Dispatch Example.

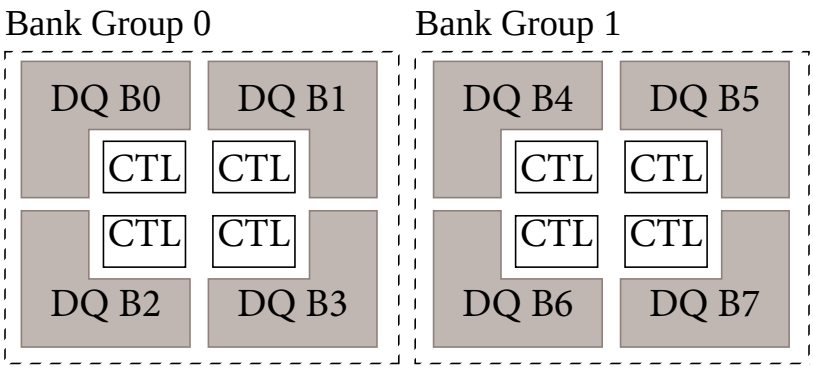


FIGURE 5-5. Two-group (eight-bank) conceptual DQ floorplan.

ARF). Dispatch reads the ARF to obtain `R1`'s value, writes both operands into the DQ, and issues the `ld` immediately. When the `add` is decoded, it consults the RCT and finds that `R3`'s previous use was as the `ld`'s destination field, and thus Dispatch updates the pointer from `ld`'s destination to the `add`'s first source operand. Like the `ld`, the `add`'s immediate operand (`55`) is written into the DQ at dispatch. Dispatching the `add` also reads the `ld`'s result empty/full bit. Had the `ld`'s value been present in the DQ, the dispatch of the `add` would stall while reading the value array.

The `mult`'s decode consults the RCT, and discovers that both operands, `R3` and `R4`, are not yet available and were last referenced by the `add`'s source 1 operand and the `add`'s destination operand, respectively. Dispatch of the `mult` therefore checks for available results in both the `add`'s source 1 value array and destination value array, and appends the `mult` to `R3`'s and `R4`'s pointer chains. Finally, like the `add`, the `sub` appends itself to the `R3` pointer chain, and writes its dispatch-time ready operand (`66`) into the DQ.

Disaggregated DQ Implementation. DQ entry allocation and dependence annotation place distinct requirements on the underlying hardware. DQ allocation is contiguous and requires substantial bandwidth. In particular, instruction metadata (approximately 12 bits) and operand values (64 bits per value) must be transferred from the dispatch logic to the destination bank group via a

dispatch bus. To optimize dependence annotation (described below), the bus also carries $2 \cdot Width \cdot (\lceil \log_2 N_{DQEntries} \rceil + 2)$ bits describing inter-instruction register dependences. A four-wide dispatch pipeline for a 1024-entry DQ requires a dispatch bus width of 684 bits, accounting for signals to indicate operand validity and actual dispatch count. A comparable conventional design's pipeline width is approximately 704 bits, at the same point.

For simplicity, dispatch hardware will operate on only one bank group in a given cycle. The group active for dispatch can be signalled with a *group dispatch enable* line. Figure 5-6 illustrates the dispatch bus, routed to each bank group. The bus is driven only by the dispatch logic and is ignored by bank groups for which the *group dispatch enable* line is not asserted.

The other function of the dispatch logic, dependence annotation, requires pointer-write operations to predecessor instructions in the DQ to establish forward dependence pointers. Commonly, instructions dispatch to the same bank group as their most recent predecessor (in fact, very often dependant instructions dispatch concurrently). These dependences are communicated to the dispatching DQ bank group though additional signalling on the dispatch bus via pointer pairs, enumerating predecessors and their successor operands. However, in the general case, instructions

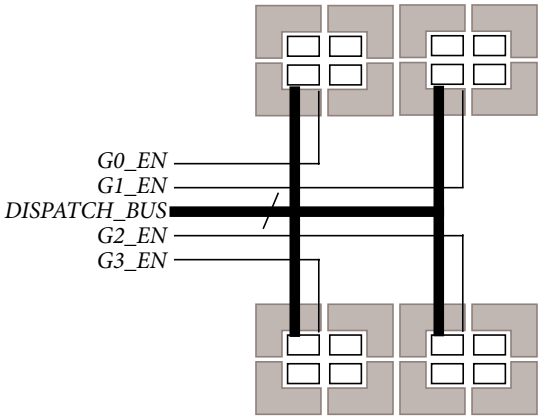


FIGURE 5-6. Dispatch Bus over Four Bank Groups.

may depend on *any* prior instruction in the DQ, not only those instructions previously dispatched to the bank group. To facilitate dependence annotation between instructions in separate groups, the dispatch logic encodes an explicit *pointer write* message onto the Forwardflow operand network. This message traverses the inter-bank-group operand network in a manner similar to that of a wakeup message (Section 5.3.8).

Upon receipt of a pointer-write message, update hardware actually performs the write of the DQ pointer array. Whenever a pointer field is written, the owning bank group forwards the value of the local operand, if the value array's empty/full bit is set. This facilitates forwarding of values from already-completed instructions to newly dispatched operations. The forwarding operation proceeds independently of DQ entry allocation, allowing both operations to execute concurrently and without ordering constraints.

5.3.3 Pipelined Wakeup

Once an instruction has been dispatched into the DQ, the instruction waits until its unavailable source operands are delivered by the execution management logic. Each instruction's DQ entry number (i.e., its address in the RAM) accompanies the instruction though the execution pipeline. When an instruction nears completion, pointer chasing hardware reads the instruction's destination value pointer. This pointer defines the value chain for the result value, and, in a distributed manner, locations of all successors through transitive pointer chasing. The complete traversal of a chain is a multi-cycle operation, and successors beyond the first will wakeup (and potentially issue) with delay linearly proportional to their position in the successor chain.

The wakeup process is illustrated in Figure 5-7. Upon completion of the `ld`, the memory value (`99`) is written into the DQ, and the `ld`'s destination pointer is followed to the first successor, the

add. Whenever a pointer is followed to a new DQ entry, available source operands and instruction metadata are read speculatively, anticipating that the arriving value will enable the current instruction to issue (a common case [95]). Thus, in the next cycle, the add's metadata and source 2 value are read, and, coupled with the arriving value of 99, the add may now be issued. Concurrently, the update hardware reads the add's source 1 pointer, discovering the mult as the next successor.

As with the add, the mult's metadata, other source operand, and next pointer field are read. In this case, the source 1 operand is unavailable, and the mult will issue at a later time (when the add's destination pointer chain is traversed). Finally, following the mult's source 2 pointer to the sub delivers 99 to the sub's first operand, enabling the sub to issue. At this point, a NULL pointer is discovered at the sub instruction, indicating the end of the value chain.

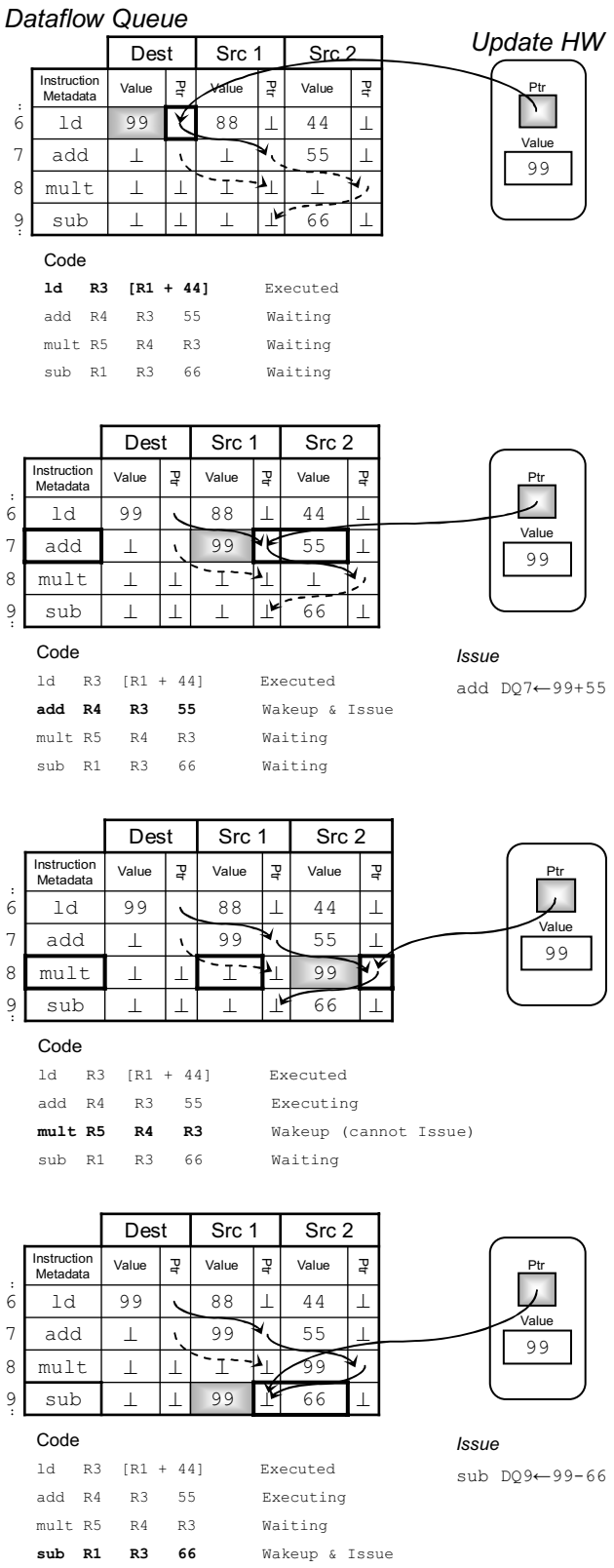


FIGURE 5-7. Wakeup Example.

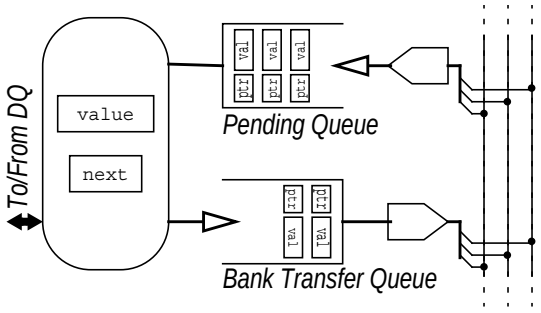
Risk of Deadlock. As stated above, the pointer chasing hardware is responsible for issuing instructions to functional units during traversal. Should a particular instruction be unable to issue because of a structural hazard (i.e., all functional units are busy), the pointer chase must stall until the instruction can issue normally. Nominally, this condition is only a minor performance overhead. Rarely, a second structural hazard can arise when a pointer chain that would normally begin its chase requires the use of stalled pointer-chasing control circuitry. This forms a circular dependence, as the functional unit cannot accept a new operation (i.e., the current result must first be collected from the functional unit) and the pointer-chasing hardware must stall until it can issue the current instruction, resulting in deadlock. The intersection of these two control hazards is rare, and can be made rarer still by modest of buffering instructions or results at functional pipelines (in practice, 14-entry buffers prevent all deadlocks). Should deadlock still arise, the circular dependence is easily detected (i.e., a local functional unit is stalled and the update hardware is stalled), and can be resolved with a pipeline flush. Ordering properties of functional units guarantee forward progress of at least one instruction after a pipeline squash.

Banking the DQ. Wakeup logic is associated with each bank of the DQ. Briefly, the DQ is sub-banked on low-order bits of the DQ entry number to support concurrent access to contiguous elements. Sub-banking delivers ample bandwidth to dispatch and commit logic—which access the DQ contiguously—without adding additional ports. Each field of the DQ is implemented as a separate SRAM (e.g., value fields are separate from each pointer field, etc.), to enable greater concurrency in the DQ management logic.

Each bank of the DQ is serviced by an independent instance of the pointer chasing hardware shown in Figure 5-8, consisting of a ***next*** pointer register, a current ***value*** register, a *pending*

queue of pointer/value pairs, and buffered ports to the interconnect between the banks of the DQ. The behavior of the wakeup logic is described in the accompanying algorithm, which runs every cycle. Since DQ entry numbers accompany instructions through functional pipelines, pointers to destination fields can be inferred as instructions complete execution.

During a given cycle, the update hardware for a particular bank will attempt to follow exactly one pointer. If no pointer is available (line 8), the DQ is not accessed by the update hardware, thereby conserving power. Otherwise, if `next` designates a non-destination field (i.e., one of the two source operands of a successor operation), the remaining source operand (if present) and instruction opcode are read from the DQ, and the instruction is passed to issue arbitration (line 15). If arbitration for issue fails, the update hardware stalls on the current `next` pointer and will issue again on the following cycle.




```
1    // Handle pending queue
2    if next == NULL:
3       next = in.ptr
4       value = in.val
5       in.pop()
6
7    if next == NULL:
8       return // No work to do
9
10   // Try to issue, if possible
11   if type(next) != Dest &&
12      dq[next].otherval.isPresent:
13      val2 = dq[next].otherval
14      opcode = dq[next].meta
15      if !Issue(opcode, val, val2):
16         return // Stall
17
18   dq[next].val = value
19   next = dq[next].ptr
20
21   // Handle DQ bank transfer
22   if bank(next) != bank(this):
23      out.push(next,value)
24      next = NULL
```

FIGURE 5-8. Pointer Chasing Hardware and Algorithm.

The update hardware writes the arriving value into the DQ (line 18) and reads the pointer at `next` (line 19), following the list to the next successor. If the pointer designates a DQ entry assigned to a different bank, the pair <`next`,`value`> is placed in the bank transfer queue

(line 23). This queue empties into an intra-bank-group crossbar, from which pointer/value pairs are routed to other banks, or to the inter-bank-group router, if necessary.

5.3.4 Commit Pipeline

Forwardflow implements a pipelined commit operation, in order to resolve writeback races to the ARF and implement speculative disambiguation (the latter is described in Section 5.3.6).

After instructions have been executed (i.e., when the empty/full bit on the destination operand's field has been set), instructions are removed from the head of the DQ and committed in program order. Commit logic removes the head instruction from the DQ by updating the DQ's head pointer and writes to the ARF where applicable. If the RCT's last writer field matches the committing DQ entry, the RCT's busy bit is cleared and subsequent successors may read the value directly from the ARF. The commit logic is not on the critical path of instruction execution, and the write to the ARF is not timing critical as long as space is not needed in the DQ for instruction dispatch.

Decode and dispatch are temporally decoupled in the Forwardflow pipeline, potentially by many cycles, which leads to a vulnerability window between the read of the RCT and a subsequent write to the ARF. As a result, there is a risk that a committing producer may write the ARF and leave the DQ *after* a successor has read the RCT but *before* the successor can establish a forward pointer from the producer. In other words, it is possible for the dependence information read from the RCT to become stale after it is read. To prevent this case, Forwardflow implements a two-phase pipelined commit, in which predecessors retain their DQ entries until dependent decoded instructions successfully establish their dependences on the committing operation.

Two-phase pipelined commit proceeds as follows. To begin, only one bank group constituting the DQ may commit in a given cycle (this bank group effectively holds a commit token). Use of a commit token does not substantially reduce performance because instructions are contiguous in program order within a bank group. Consequently, a bank group can trivially predict when to relinquish the token as well as the next bank group requiring the token.

In a given cycle, the bank group holding the commit token may elect to begin retirement of some number of contiguous instructions, depending on the number of local instructions that have completed execution. The committing group communicates the architectural register names, DQ entry numbers, and produced values of the retiring instructions to the ARF, via a dedicated interconnect (this is implemented as a circuit-switched bus between all possible bank groups and the ARF control logic). Pipelining arises naturally from his procedure, as the ARF may not reside within single-cycle communication distance, depending on which bank group currently holds the token. The committing group does not recycle the DQ space associated with committing instructions until an acknowledgement has been received from the ARF (which resides in the centralized dispatch logic). This acknowledgement is ordered with respect to instructions that had already passed the RCT upon the commit request's receipt at the ARF. Therefore, the committing group may infer that, upon receipt of the acknowledgement signal, all subsequent instructions that may have sent a pointer write operation to the committing DQ entries have already completed their dispatch operations.

Note that pipelining this operation allows the bank group to commit at full bandwidth, with depth determined by the round-trip latency between the bank group and the ARF.

5.3.5 Control Misprediction

Like other out-of-order machines, Forwardflow relies on dynamic branch and target prediction to increase lookahead. The misprediction recovery strategy used in this work is a checkpoint-restore approach, inspired by checkpoint-based processing techniques [155, 3, 190], in which a snapshot (i.e., a checkpoint) of microarchitectural state is recorded at speculation points. This checkpoint can be restored at a later time if the associated control transfer instruction is discovered to have been a misprediction, thereby restoring the checkpointed pipeline structure to an earlier state without explicitly walking or scanning any (potentially large) structures. Prior work in checkpoint-based architectures has evaluated techniques to reduce checkpoint space and time overheads through aggressive checkpoint recycling, coarse-grain speculation, and other techniques. However, checkpoint optimization is not a foci of this work, and I made no effort to innovate in this space, nor to significantly optimize checkpoint overheads. Overall, the power and area overheads of unoptimized checkpoints are manageable.

However, not all forms of recovery can be handled by restoring from checkpoints. When exceptions, interrupts, or disambiguation mispredictions occur, correct recovery is implemented as a flush of all instructions that follow the excepting instruction. Since these events are handled at commit-time, the flush constitutes the entire instruction window (i.e., selective squash is not necessary in this case). After a complete flush, decode resumes with an empty RCT, as no instructions are in-flight. The performance impact of these squashes is determined by their frequency: so long as they are sufficiently rare, they will not constitute a performance bottleneck. However, prior work [183], as well as my own empirical observations, indicate that the effect of these flushes can be substantial in the target ISA (SPARCv9) due to frequency of register window and TLB traps

(these traps are odious to any ILP-intensive implementation of SPARCv9). This observation motivated hardware support for trap handling, discussed in Appendix A.

For all checkpointed state in Forwardflow, it is assumed that checkpointing logic is optimized for fast checkpoint creation, as this operation is on the critical path of branch dispatch. Checkpoint *restoration* is not highly optimized, and is a fairly expensive operation. Dispatch of new instructions cannot resume until all checkpoints have been properly restored. However, so long as checkpoint restoration time is less than the combined latencies of the frontend pipeline and L1-I cache, checkpoint restoration time plays no role in performance, as it incurs no *additional* overhead.

In Forwardflow pipelines, three key operations underlie correct misprediction recovery. In particular, recovery mechanisms must:

- Restore the RCT's state as it was before the instructions following the branch were decoded,
- Invalidate all false-path instructions, ensuring that no false-path instructions ever update architectural state, and,
- Ensure no values generated on the mispredicted path ever affect true-path instructions.

Restoration of RCT state is accomplished by checkpointing the RCT on predicted branches, a technique identical to the checkpointing of a register rename table [190]. This work assumes that the RCT is specifically architected at the circuit level to accommodate this checkpoint capture/restore operation, and that checkpoints themselves reside in a flattened SRAM array. This array need not have single-cycle access time, so long as writes to the array can be pipelined. Restorations of RCT checkpoints from the array are the most performance-critical checkpoint-restore operation in the microarchitecture, and must be accomplished no slower than the L1-I access latency, in

order to keep RCT recovery off of the critical path of misspeculation recovery. Estimates of array access times from CACTI corroborate this expectation as feasible.

Invalidation of false-path instructions is trivial in the Forwardflow microarchitecture. Like other microarchitectures, the front-end pipeline is completely (i.e., non-selectively) squashed on any misprediction. Because the DQ is managed as a FIFO, simple manipulation of the FIFO's tail register is sufficient to reclaim DQ space occupied by false-path instructions, effectively invalidating the instructions themselves and ensuring they will not commit (i.e., the DQ's tail register is set to the DQ index following the branch instruction, modulo the DQ size). In SPARCv9, this carries the consequence of also flushing the branch delay slot, but the benefit of salvaging the delay slot does not motivate additional complexity to do so (the delay slot can, for instance, modify registers, which would affect the timing of checkpoint capture mechanisms in earlier pipe stages).

The insulation of all true-path *instructions* from false-path *values*, is more subtle than the previous two requirements. Because Forwardflow has no mechanism to re-schedule cancelled instructions, it is not feasible to squash functional pipelines on mispredictions (in the general case—some exceptions exist and are described subsequently). Instead, functional pipelines must be allowed to drain normally (i.e., quiesced), and values belonging to false-path instructions must be discarded as part of the recovery process. This prevents any previously-scheduled false-path instruction from propagating its value to later true-path instructions, which otherwise would occur due to DQ space reclamation.

Lastly, in order to cancel dependences already established at misprediction-time from true-path instructions to false-path instructions, we augment the pointer fields with valid bits, stored in a separate flip-flop-based array. Pointer valid bits are checkpointed on control predictions, as was

the case with the RCT. This operation was originally proposed for use in pointer-based schedulers [136], and is not a novel contribution of this work.

Single Bank Group Branch Misprediction Procedure. As previously described, Forwardflow's execution logic is organized into DQ bank groups, each backing a subset of the DQ's aggregate storage space, integer, floating point, and memory datapaths, and associated control logic. Bank group size is bounded largely by wire and logic delay—a group defines the single-cycle communication boundary. Small Forwardflow cores may contain only one bank group; larger cores may encompass several bank groups.

When mispredictions occur during single-group operation, recovery is initiated as previously described: functional pipes are quiesced, checkpoints are restored as appropriate, and the tail pointer is reset to the instruction following the misprediction event. This operation requires no communication with other bank groups, and the recovery process described above is sufficient to restore proper pipeline operation.

Branch Misprediction Spanning Multiple Bank Groups. Larger Forwardflow cores encompassing multiple bank groups require additional hardware to support inter-group control misspeculation. In particular, bank groups must notify one another of misspeculations. Consider a Forwardflow core consisting of many bank groups. Suppose bank group A detects a misprediction event of control instruction I (note that I must reside in A). With respect to a given misprediction, for all possible A, bank group B relates to A in one of four ways:

- **Local**: When $B = A$, the misprediction event is local to bank group B.

- **Downstream**: *B* is considered to be downstream of a *A* (and therefore, downstream of *I*) when all instructions currently resident in *B* follow the misprediction event in the previously-predicted program order.
- **Upstream**: *B* is considered to be upstream of a *A* (and therefore, upstream of *I*) when all instructions currently resident in *B* precede the misprediction event in the previously-predicted program order.
- **Mixed Stream**: *B* is neither upstream nor downstream of *A* (nor is *B* local), because *B* contains both the DQ's head and the DQ's tail (i.e., the DQ is nearly full).

Note that it is always possible for *B* to identify its relationship to *A*, because state local to *B* determines the head and tail pointers, the immediate (static) downstream and upstream groups (i.e., determined by core floorplan), and obviously *B* can detect the $B = A$ case (Local).

Recovery actions from the misprediction vary according to *B*'s relation to *A*.

- **Local**: $B = A$ follows the procedure outlined above to recover from a single-bank-group misprediction. The local bank group still informs all other bank groups of the misprediction, as other bank groups must restore still pointer valid bits. The local bank group assumes control of dispatch, which will resume when instructions are available from the frontend.
- **Downstream**: All instructions in *B* are false-path instructions. Therefore, *B* invalidates all local instructions. Pipelines can be squashed if desired, or quiesced as normal, discarding all results. If *B* currently controls dispatch, this control is relinquished to *A*.

- **Upstream**: No instructions in *B* are false-path with respect to *I*. Therefore, *B* need not invalidate any instructions, but *B* must restore its checkpoint of local pointer valid bits, as *B* may still contain pointers to the invalid region following *I*. Local pipelines need neither be squashed (or quiesced).
- **Mixed Stream**: *B* invalidates all instructions preceding the DQ head pointer. Local pipelines must be quiesced. *B* restores the appropriate checkpoint of local pointer valid bits.

Because bank groups in general do not reside within single-cycle communication range of other groups, it is possible for more than one branch misprediction to be detected in a single cycle. However, only one such misprediction is the eldest, and because groups are able to ascertain upstream/downstream relationships with other groups, they are similarly able to identify the eldest misprediction. Of course, recovery must always reflect the eldest misprediction in flight. Note that the actions detailed in this section for recovery from a misprediction never precludes a bank group's ability to recover from misprediction of an elder branch, which may be dynamically resolved in subsequent cycles.

5.3.6 Speculative Disambiguation

Forwardflow leverages NoSQ [146] to implement memory disambiguation. The general operation of NoSQ in the context of an SSR-based core is detailed in Chapter 4—briefly, store-to-load dependences are represented with inter-instruction pointers, as though they were register dependences. The NoSQ set-associative dependence predictor is consulted at decode time: stores predicted to forward record their DQ entry numbers in the NoSQ hardware. Subsequent loads predicted to forward from a particular store establish a dependence on this write operation. The

store-to-load dependence is then resolved as a phantom register dependence by the Forwardflow pipeline.

However, the speculative nature of NoSQ requires verification of speculative disambiguation. This is accomplished in two ways, based on Sha et al. First, a store sequence bloom filter (SSBF) records the store sequence number (SSN) of the last store operation to write the subset of the address space represented by each SSBF entry. All loads consult the SSBF to verify correct forwarding. NoSQ annotates dependent loads with the SSN from which they forward. Loads that forwarded from earlier stores compare their recorded SSNs to those in the SSBF: a mismatch indicates a possible misprediction (i.e., a possible NoSQ false positive, a predicted forwarding that did not occur). Loads speculated as independent are guaranteed to be independent of earlier in-flight stores, if they observe an SSN in the SSBF lower than the SSN observed by the load at dispatch-time (i.e., $SSN_{Dispatch}$—this data is retained by the NoSQ hardware). If the SSN retained in the SSBF is larger than the load's $SSN_{Dispatch}$, a misprediction is possible (i.e., a NoSQ false negative, the failure to predict a store-to-load forward event).

When the SSBF indicates a possible mispredict (either false-negative or false-positive), the committing bank group forced to replay the vulnerable load in the L1-D. The value originally observed by the load is compared to the value observed by the load's replay—if the values differ, a misspeculation is triggered and the pipeline is flushed.

In order to handle races with remote threads concurrently accessing a shared block, the SSBF is updated with a sentinel value whenever a cache line is invalidated in the L1-D due to a remote exclusive access request. A load observing the sentinel value in the SSBF at commit-time *always* replays, thereby preventing a load from ever committing a value not allowable under the memory

consistency model (in this work: sequential consistency). Replayed loads are instead forced to observe the correctly-sequenced value from memory. This mechanism overall is conservative of when to require replays (e.g., due to false sharing and SSBF conflicts), but does not squash unnecessarily (hence the value comparison). The use of a sentinel in the SSBF represents a small extension to NoSQ to make it suitable for multiprocessors.

5.3.7 Bank Group Internal Organization

DQ bank groups implement a contiguous subsection of DQ space. Therefore, a DQ bank group must match dispatch bandwidth with that of the frontend pipeline (peak 4 instructions per cycle). Within a bank group, instructions can be interleaved among separate banks (thereby interleaving DQ space on low-order bits within a bank group), or dispatched to a single structure. The former approach places constraints on the wakeup and execution logic (i.e., the logic must operate across independent banks), and the latter requires multiple ports on a single RAM, increasing its area, power, and delay. Furthermore, execution and commit logic will place additional demand on the DQ bank groups—both roughly comparable to the throughput demands of dispatch logic.

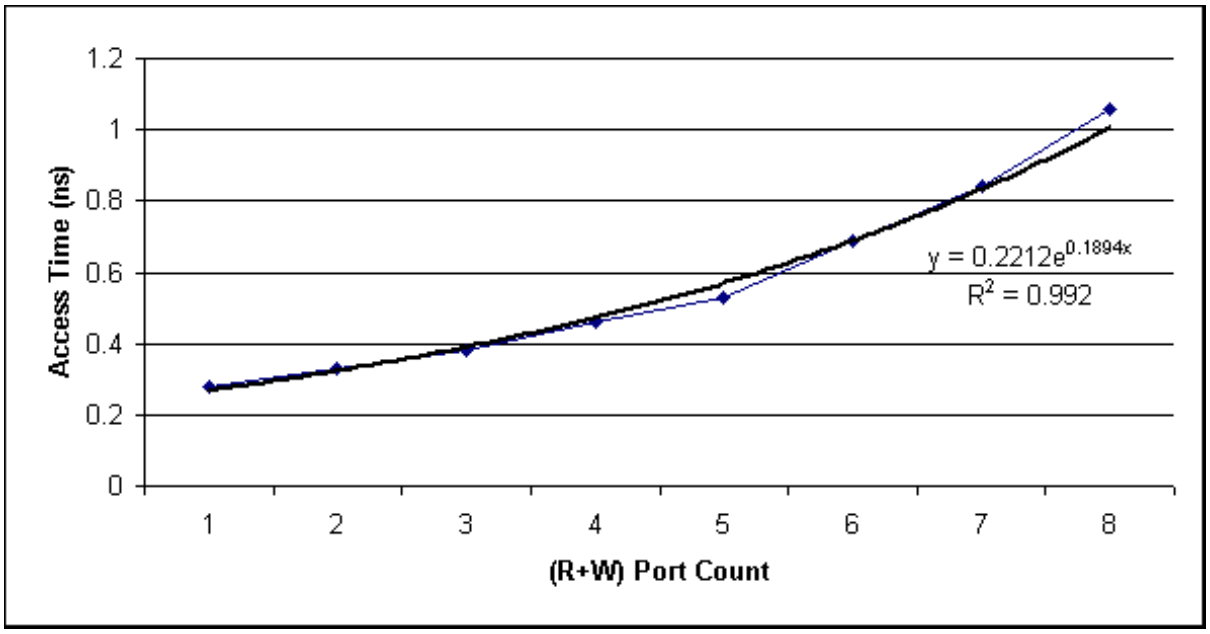


FIGURE 5-9. Access time (*ns*) versus port count for a single value array in a DQ bank group.

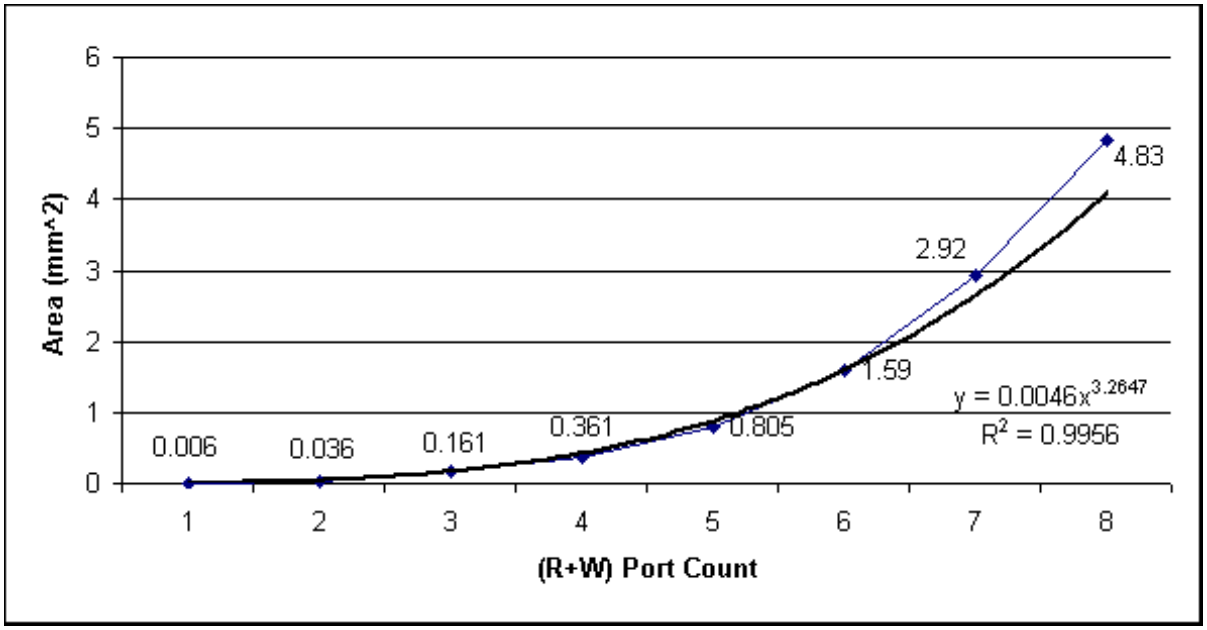


FIGURE 5-10. Area (mm^2) versus port count for a single value array in a DQ bank group.

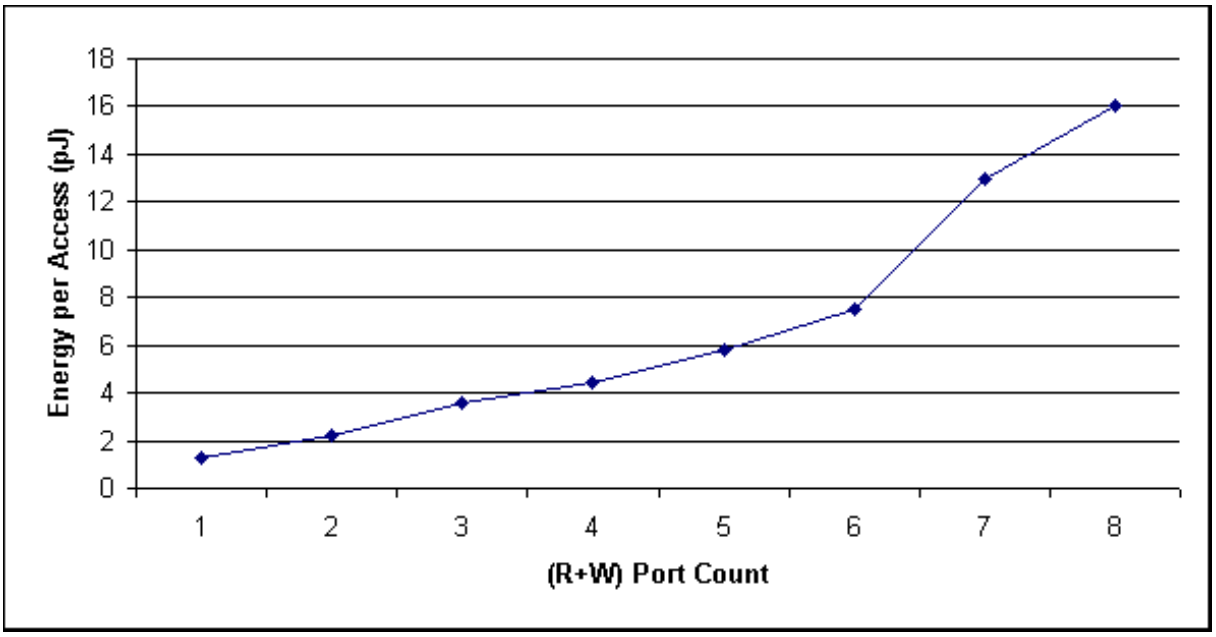


FIGURE 5-11. Energy per access (*pJ*) versus port count for a single value array in a DQ bank group.

To more fully understand how these overheads scale in port count, Figure 5-10 plots area versus port count for the value array component of a DQ bank group of size 128 (details of evaluation can be found in Chapter 3—briefly, these estimates are derived from CACTI). Further, Figures 5-9 and 5-11 plot access time and energy consumption, respectively. Trend lines are provided where $R^2 > 0.9$.

Area (Figure 5-10) grows (approximately) with the cube of ports on the array. This trend is due to the need for internal duplication of data arrays to meet growing port demand (the actual degree of duplication required is technology-dependent). As area grows, access time (Figure 5-9) grows *exponentially* in the port count. This accounts for internal routing within the multi-ported array, requiring successively longer RC-dominated wires to traverse the duplicate arrays, and to implement write-to-read port bypassing. Access energy (Figure 5-11) is approximately piecewise linear in the number of ports. Therefore, the power consumed by multi-ported arrays is at least quadratic

in the number of ports (i.e., energy consumed in one cycle, $E_{cycle} = n_{access} \cdot E_{access}$. The number of accesses, n_{access} , is linear in port count; E_{access} is also linear). Together, these trends argue for low port counts on high-performance SRAM arrays, like those used in the DQ bank groups.

Area affects not only the access time of the DQ, but also wire delay within a DQ bank group, and between bank groups. Figure 5-12 plots the wire delay (assuming optimal repeater placement) to traverse a linear distance equal to the square-root of the corresponding array area—the hypothetical delay of a wire traversing the array, assuming an aspect ratio of one. This delay and the DQ access time are a rough bound on cycle time, assuming little or no logic delay.

These trends make a case for interleaved bank groups. Interleaving enables a single port to be used for both dispatch and commit logic, as both of these operations operate on contiguous DQ entries: dispatch and commit accesses are guaranteed to access separate banks if interleaved on low-order bits of the SRAM address (DQ entry number). Dispatch and commit logic can easily

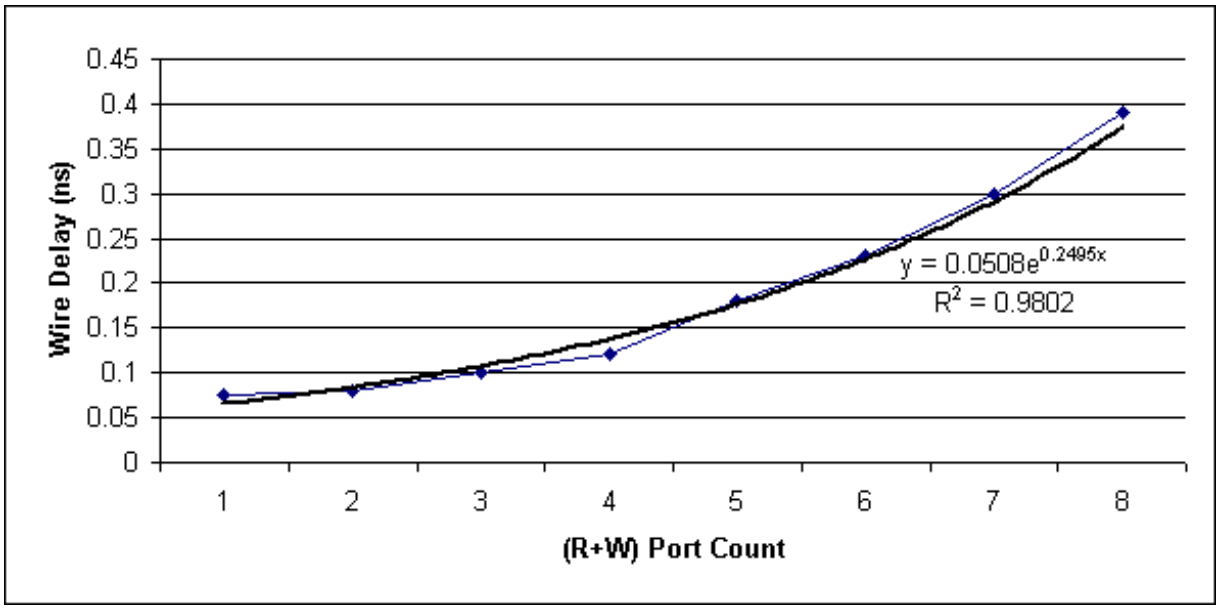


FIGURE 5-12. Wire delay (*ns*) versus port count to cross a DQ value array.

arbitrate for use of ports: if different bank groups are active for dispatch and commit, both can observe full bandwidth and use all banks. When dispatch and commit target the same group, each predictably stride through the DQ space, allowing these processes to share banks in an alternating pattern, switching port ownership predictably on each cycle.

DQ wakeup operations are less predictable than dispatch and commit. The behavior of wakeup is determined by inter-instruction data dependences rather than adjacency of instructions in predicted program order. To accommodate the unpredictable access patterns of wakeup accesses, a port is dedicated to these accesses. Whereas arbitration between commit and dispatch logic is straightforward, it is difficult to predict when update hardware will access the DQ. Hence, each bank within a group has a port dedicated only to the update hardware.

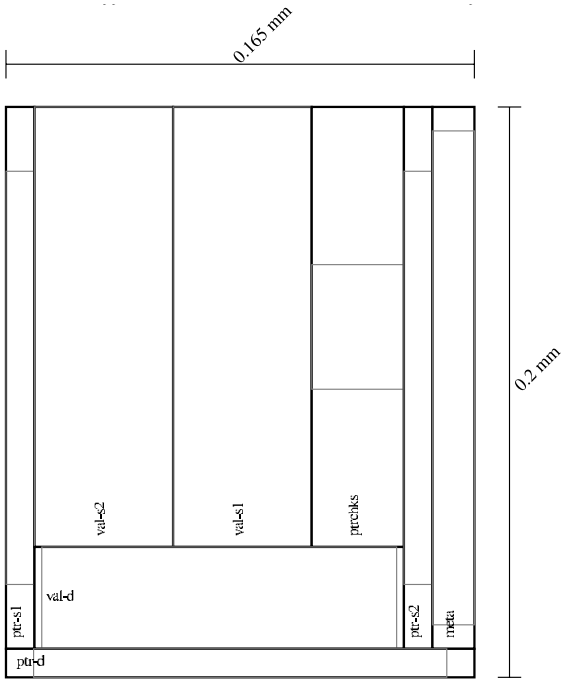


FIGURE 5-13. DQ bank consisting of 32 entries and valid bit checkpoints.

Assigning a single port to dispatch/commit and a second port to update hardware, DQ structures are only two-ported. In other words, the actual devices used in the implementation represent the lowest ends of the trends presented in the figures above—moreover, four-way bank interleaving (i.e., the degree of interleaving to service a frontend of width four) implies that the actual size of the individual DQ banks is even smaller than those used in the figures above (though larger than one quarter size, as SRAM sense amplifiers tend not to scale linearly in array size).

Assuming an overall two-ported, four-way interleaved design, Figure 5-13 shows a floorplan of a single bank of the DQ, decomposed into seven DQ fields (three 64-bit value arrays, three 12-bit pointer arrays, a 16-bit metadata array) and a 256-entry checkpoint array for pointer valid bits. The total area of a DQ bank with the above design requirements is 0.033 mm^2. Manhattan distance is 0.36 *mm*, and each bank has an access time of approximately 90 *ps* in 32 nm technology.

Using Figure 5-13's plot of a DQ bank layout, Figure 5-14 plots a complete floorplan of a DQ bank group. The bank group design is 0.87 mm^2 and has an internal area efficiency of 87%. Maxi-

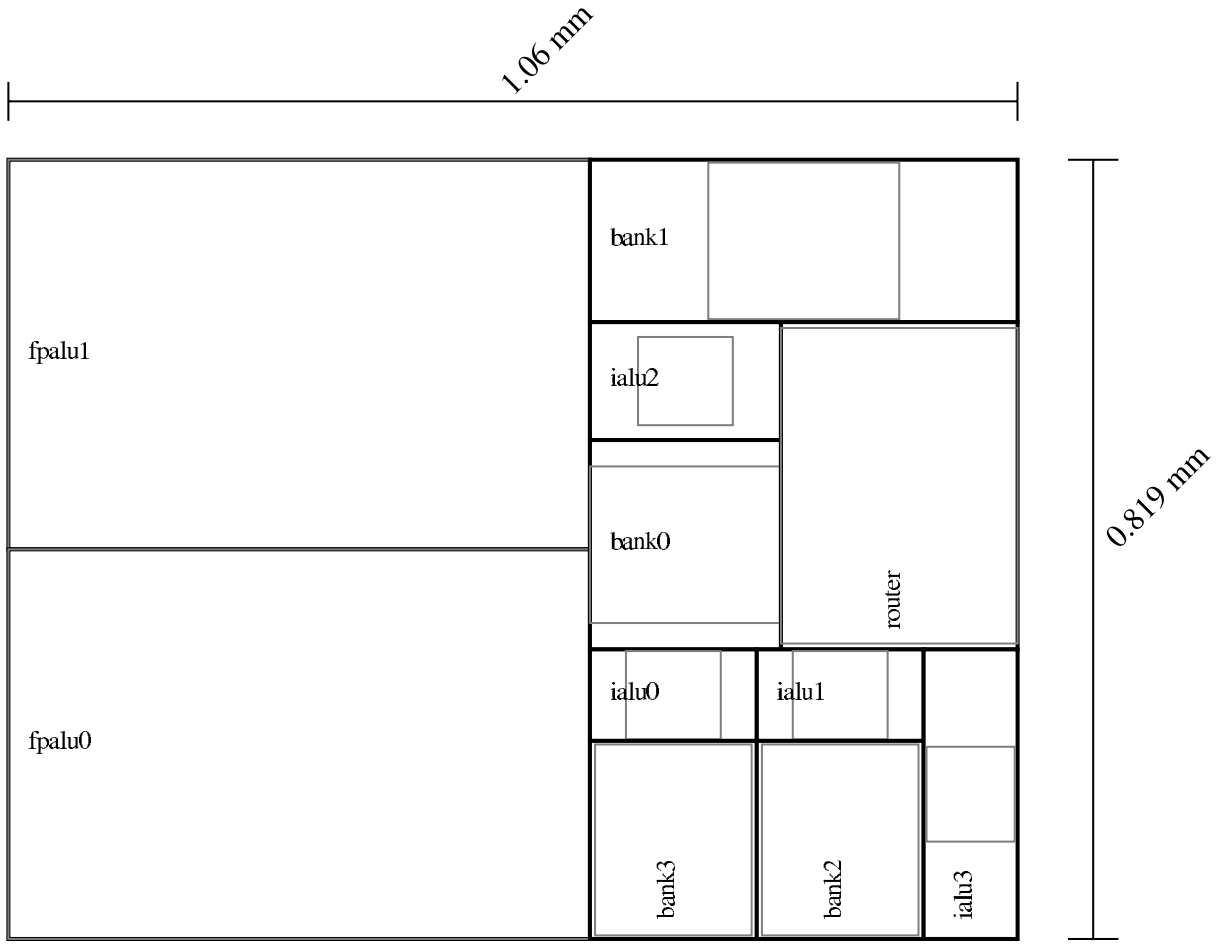


FIGURE 5-14. Floorplan of a 128-entry DQ bank group for use with a four-wide frontend.

mum Manhattan distance is 1.2 *mm* between banks. I estimate maximum intra-group signalling delay at 0.24 *ns*.

Within a bank group, the logic implementing the bank transfer logic consists of a five-way crossbar, 72-bits wide. The inter-group router and individual DQ banks are endpoints on the crossbar. The router is responsible for sending non-local pointer/value pairs to other bank groups, and for delivering pointer/value pairs and pointer write operations to the local bank group.

5.3.8 Operand Network

A dedicated scalar operand network routes pointer/value pairs and pointer write messages (used in dispatch, Section 5.3.2) between DQ bank groups. Abstractly, the role of this interconnect is to deliver small data payloads (i.e., 64-bits) to a DQ bank group corresponding to a particular data pointer.

Intuition suggests that intra-bank group communication would be performance-critical: trivial experimentation shows this to indeed be the case. The importance of *inter*-bank group communication, facilitated by the scalar operand network, is less clear. If one assumes *pointer distance*—the number of instructions between endpoints of a given pointer—is independent of DQ entry number, some insight can be gained by an analysis of pointer distance. Figure 5-15 plots the cumulative distribution function of pointer distance of three typical benchmarks: `astar` and `sjeng` from SPEC INT 2006, and `jbb` from the Wisconsin Commercial Workload Suite. Other benchmarks typically exhibit similar behavior.

Overall, pointer distance tends to be short. Over 90% of pointers have a distance of eight or fewer, indicating that producers and successors tend to dispatch near one another in the DQ at

run-time. Nearly *all* pointers (greater than 95%) have a pointer distance less than or equal to sixteen (one eighth the size of a DQ bank group). The general trend of short pointers suggests that the utilization of the operand network will be low under most workloads, as most data dependences are local.

Experimentation reveals this intuition to be accurate. Among the 33 single-threaded benchmarks used in this chapter's evaluation, there is no significant performance loss incurred from use of a single-cycle unidirectional ring interconnect, as compared to a perfect, zero-latency inter-bank group interconnect (i.e., performance differences are easily within the margin of error of the evaluation methodology). Performance differences become significant for more latent interconnects, but given the size of a DQ bank group (i.e., 0.87 mm^2), inter-group latency should be small.

The fact that a simple, unidirectional ring topology works nearly as well as an idealized interconnect results from the distribution of traffic classes within a Forwardflow core. Communication

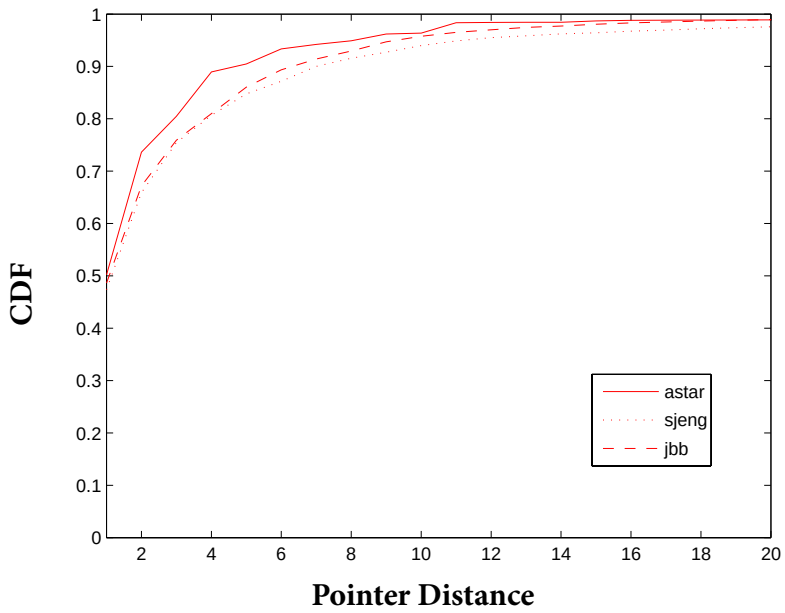


FIGURE 5-15. CDF of pointer distance for three representative benchmarks.

within a single bank group is, by far, the most common class of traffic, arising from pointers between instructions within the same bank group (about 95% of all traffic). Such traffic traverses only the local crossbars within a bank group—it does not rely on the *inter-bank-group* operand network at all. Of the remaining traffic, about 3% consists either of pointer write operations, which are fairly insensitive to latency, as they only affect the commit pipeline. Pointers crossing from one bank group to the next account for about 2% of traffic, and pointers spanning multiple bank groups are rare (less than 1%).

Given that Forwardflow cores operate well with simple operand networks (unidirectional ring), I did not investigate other topologies. After initial experiments, it became clear that the principal area of gain in operand networks was to simplify resource requirements, by employing fewer wires (e.g., with fewer links in the topology) and simpler routers (e.g., three-port ring-based routing). A ring already comes close to minimizing the number of required links in a connected network, and requires only the simplest of routers. Further exploration of this area of the microarchitecture does not seem warranted, given the small potential gains to be made.

5.4 Comparison to Other Core Microarchitectures

The evaluation of the Forwardflow microarchitecture focuses on four main areas. First, Forwardflow aims to deliver improved performance through greater exploitation of MLP. Intuition suggests that, in the shadow of a load miss, there is ample time available to discover and execute independent operations. This added parallelism should overcome the overheads of serial wakeup and resource conflicts, and deliver increased performance at larger Forwardflow DQ sizes. To evaluate the effectiveness of the Forwardflow window overall, I compare it against a hypothetical,

idealized RUU-like machine (*RUU* from Chapter 3), and consider how well each of several Forwardflow designs exploits ILP and MLP, compared to this baseline.

Second, the Forwardflow microarchitecture exhibits some improvements over a canonical out-of-order microarchitecture, but also suffers from some drawbacks as well. To justify a new core design, a Forwardflow core should, at least, exceed a traditional microachitecture's performance at the same power, or match its performance at lower power. The second area of evaluation considers how the Forwardflow design compares to a traditional out-of-order data point, with respect to performance, power, and efficiency metrics.

Third, intuition and prior work suggest that greater performance will be delivered through exploitation of larger instruction windows. To better exploit ILP and MLP, designers should build the largest possible window for a given power budget. I present experiments detailing the power/performance tradeoffs of Forwardflow windows, from very small sizes (e.g., 32 entries) to very large windows (e.g., 1024 entries).

Lastly, Forwardflow is not the only core microarchitecture to explicitly emphasize MLP. The final area of evaluation addresses how a Forwardflow core compares against two previous MLP-aware microarchitectures, *Runahead* and *CFP* (cf Chapter 2 for details on these proposals, Chapter 3 for details of implementation of these designs in this study). I evaluate Forwardflow in the context of these other techniques.

A following subsection addresses each of these four areas. I present aggregated data in most cases, to focus on general trends across many individual benchmarks. For completeness, figures detailing each individual benchmark are available at the conclusion of the chapter (Section 5.4.7), preceded by brief discussion of interesting outliers (Section 5.4.5).

TABLE 5-1. Machine configurations used for quantitative evaluation.

Configuration	Description
F-32 through *F-1024*	Forwardflow processors equipped with various DQ sizes. Each DQ bank group backs 128 entries of the DQ, and includes two integer and two floating point datapaths. DQ sizes smaller than 128 use a subset of banks within the bank group, which reduces bandwidth for dispatch, issue, and commit (i.e., *F-32* is one-wide, *F-64* is two-wide).
RUU-32 through *RUU-1024*	RUU-based processors, with varied RUU size. Equipped with the same number of integer and floating point pipelines as the corresponding Forwardflow core of the same size, though issue and writeback operations are idealized.
OoO	A canonical out-of-order core with a 32-entry scheduler and a 128-entry ROB. *OoO*'s functional pipelines are equivalent to a single Forwardflow bank group (i.e., *F-128*).
CFP	An out-of-order processor equipped with deferred queues for continual flow. *CFP* uses a ROB instead of checkpoints. This ROB can buffer the same number of instructions as the largest Forwardflow configuration used (i.e., 1024 instructions).
Runahead	The *OoO* configuration equipped with Runahead Execution, including a 4KB Runahead Cache (dirty runahead values only).

Table 5-1 details the machine configurations used in this study. More details of the individual machine models can be found in Chapter 3's description of the simulation methodology.

The single-threaded benchmarks used in this study are assumed to run on one core of an 8-core CMP (cf Chapter 3). The full L3 capacity is available to the single operating cores—other cores are assumed to have been shut down to conserve power. Note that all power estimates reflect chip-wide consumption.

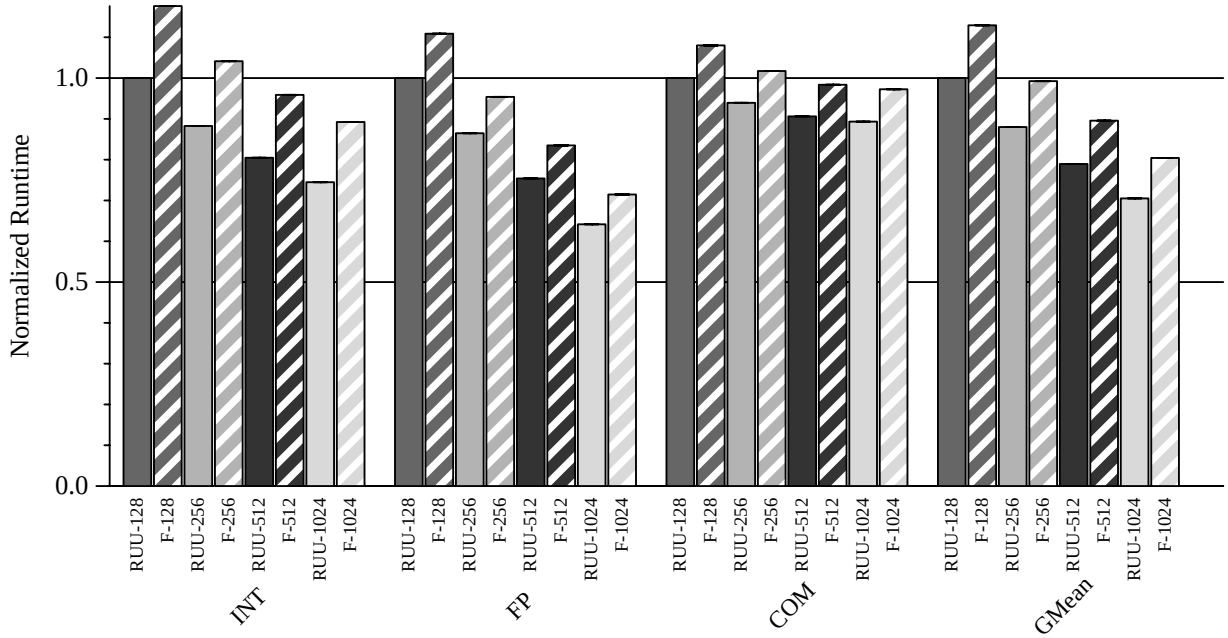


FIGURE 5-16. Normalized runtime of SPEC INT 2006 (INT), SPEC FP 2006 (FP) and Wisconsin Commercial Workloads (COM), over four window sizes.

5.4.1 Forwardflow Comparison to an Idealized Window

To begin, I frame the performance capabilities of the Forwardflow design in the context of an idealized design with respect to a variety of window sizes. I remind the reader that the effective issue width and functional unit resources of a Forwardflow design varies with the number of DQ bank groups—I vary the baselines used in this section in the same manner as those of a Forwardflow core.

The baseline used in this section is *RUU*, based on Sohi's *Register Update Unit* [153]. The complete details of *RUU* are available in Chapter 3; briefly, *RUU* can schedule any instruction in its window as soon as that instruction's dataflow dependences are satisfied. Moreover, it uses oracular issue logic to ensure minimum possible functional delay for all issued instructions, regardless of locality. In this respect, *RUU* functions as a performance upper-bound for a given window size.

This section seeks to address two general questions about the Forwardflow design. First, how closely does the Forwardflow design approach the idealized upper bound on performance? Second, to what degree does the Forwardflow core successfully exploit available MLP and ILP to improve performance? In both of these cases, comparison to *RUU* enables a limit study.

Figure 5-16 plots normalized runtime across single-threaded benchmarks from SPEC INT 2006, SPEC FP 2006, and the Wisconsin Commercial Workload suite. Runtimes are normalized to that of *RUU-128*, an *RUU*-based design utilizing a 128-entry instruction window. The rightmost stack, in this and subsequent graphs, is an average (in this case, geometric mean) over all benchmarks with equal weight (i.e., the figures do not weight each benchmark *group* equally).

Not surprisingly, performance trends better with larger window sizes, on average. Forwardflow designs trail the performance of idealized windows by 10-20% on average. I demonstrated in Chapter 4 that some performance degradation is expected with respect to an idealized window, merely from the use of SSR to represent dependences. However, this overhead is usually less than 10%. The remainder of the performance degradation observed in the Forwardflow designs arises from resource constraints within the microarchitecture itself—the effects of finite port counts, bank-interleaving, operand networks, etc.

Despite trailing *RUU*, Forwardflow performance scales with window size. The degree to which particular benchmarks scale is workload-dependent, as some workloads exhibit naturally higher or lower ILP and MLP than others—e.g., commercial workloads tend to scale less well with window size than do the integer and floating point codes. On average, each doubling of the Forwardflow aggregate DQ size reduces runtime by a further 10% (with respect to the performance of *F-128*—the effect is not compounded).

The reasons for Forwardflow's performance scaling are twofold. First, despite a slower overall window operation, Forwardflow is able to expose more than 85% of the MLP exposed by a comparable *RUU* design, as illustrated in Figure 5-17. For the purpose of this figure, I adopt a definition of memory-level parallelism similar to that suggested by Chou, Fahs, and Abraham [37]: "We (Chou, Fahs, and Abraham) define average MLP, MLP [sic] as the average number of useful long-latency off-chip accesses outstanding when there is at least one such access outstanding." I alter this definition slightly: accesses to instruction memory from the fetch pipeline are not included (arguably, the context of Chou et al. excludes instruction accesses). Unfortunately, this metric is nearly impossible to measure precisely. Instead, the method I use to approximate observed MLP tracks when committed load misses begin execution, and when they are completely serviced. Concurrent accesses to the same cache line are discarded (identified by dependence on the same MSHR). There are several minor flaws with this approach—for instance, the beneficial effects of

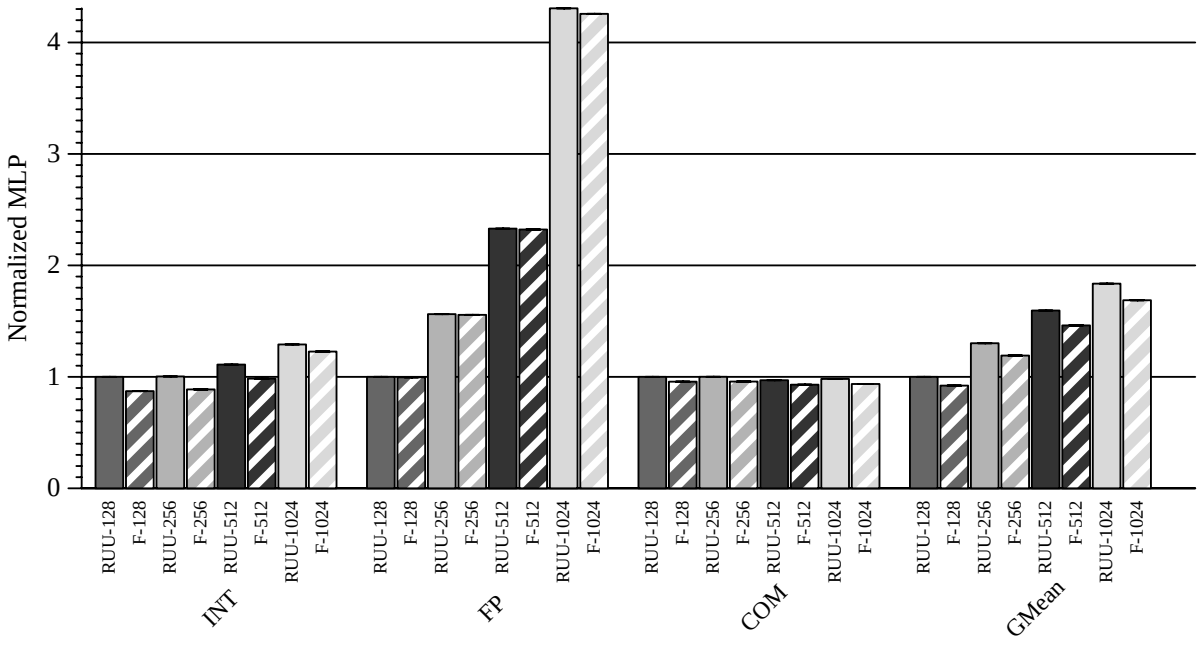


FIGURE 5-17. Normalized memory-level parallelism (MLP) of SPEC INT 2006 (INT), SPEC FP 2006 (FP) and Wisconsin Commercial Workloads (COM), over four window sizes.

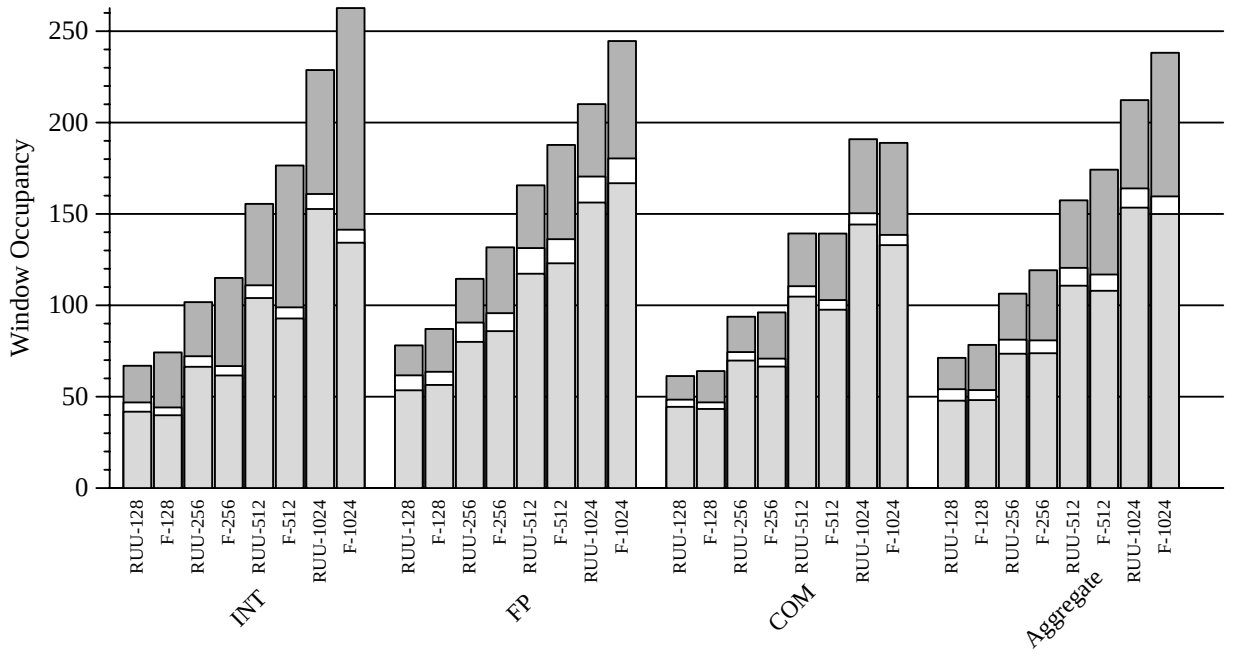


FIGURE 5-18. Categorized window occupancy (Completed (bottom), Executing (middle), and Waiting (top) instructions), SPEC INT 2006 (INT), SPEC FP 2006 (FP), and Wisconsin Commercial Workloads (COM), over four window sizes.

false-path loads are not considered, and loads executed under *Runahead* (Section 5.4.4) cannot be profiled, as they never commit. Moreover, this definition is sensitive to second-order effects, e.g., order of branch resolution determines whether loads actually execute concurrently, or not, yielding a supra-linear MLP estimate in one SPEC FP workload in the *F-1024* and *RUU-1024* cases (`GemsFDTD` exhibits 10x more estimated MLP over 128-entry windows). Overall, however, this approach seems a good approximation of the Chou et al. definition.

To gain further insight into Forwardflow's operation with respect to an idealized window, I classify in-flight instructions into three possible states (Figure 5-18). *Waiting* (top-most) instructions have dispatched, but have not yet been selected for execution: these instructions are either waiting for predecessor operations to complete, or (in the Forwardflow case only) are waiting for update hardware to traverse the appropriate dependence list. *Executing* (middle) instructions have been issued to functional pipelines or the memory subsystem, but have not yet produced an output

value (or completed execution, if they do not produce a value). Lastly, *Completed* (bottom-most) instructions have successfully executed, but have not yet committed—either an earlier non-completed instruction precedes the completed instruction in program order, or the commit logic has simply not yet reached the instruction in question.

Each of these instruction states places varying demands on the underlying microarchitecture. *Waiting* instructions must be scheduler-resident. *Executing* instructions occupy execution resources. *Completed* instructions must simply be buffered until commit. Overall, Forwardflow's occupancy for a given window size tends to be slightly larger than that of *RUU*, but *Executing* occupancy is lower on average. Both of these phenomenon occur for the same reason: Forwardflow's SSR-based value-delivery mechanisms are not instantaneous. This reduces mean *Executing* occupancy, because successors of the same value do not wake in the same cycle. This also increases mean window occupancy, by delaying branch misprediction discovery and handling (i.e., there is additional occupancy of false-path instructions).

Both Forwardflow and *RUU* expose ILP and MLP to achieve high performance. To approximate a measure of the former, I use *Executing* occupancy as a measure of the workload's underlying instruction-level parallelism[1]. The degree to which ILP and MLP predict the performance across machine configurations is presented in Table 5-2, based on linear regression of runtime with respect to ILP and MLP. Overall, the *RUU* design, with a greater degree of idealization, tends to have more predictable performance than Forwardflow, which suffers from a few performance

1. This is an approximation in several ways. First, occupancy includes the effects of false-path instructions. Consequently, the measure is subject to the accuracy of branch prediction. Second, executing occupancy includes contributions from load misses, and is therefore not independent of the measure of memory-level parallelism.

TABLE 5-2. Coefficient of determination R^2, as determined by univariate and bivariate linear regression across Forwardflow and RUU-based designs.

Machine Model	Variables Considered	Mean R^2	Median R^2
RUU	MLP	0.82	0.89
RUU	ILP	0.69	0.85
RUU	MLP and ILP	0.96	0.98
Forwardflow	MLP	0.79	0.93
Forwardflow	ILP	0.79	0.88
Forwardflow	MLP and ILP	0.95	0.97

artifacts specific to the microarchitecture (e.g., the sensitivity of `bzip2` to the effects of SSR). Overall, however, Forwardflow's performance tends to follow the available ILP and MLP.

5.4.2 Forwardflow Comparison to a Traditional Out-of-Order Core

I next discuss Forwardflow's behavior in the context of a more realistic design, a contemporary out-of-order core (*OoO)*. This section focuses only on Forwardflow configurations exhibiting similar performance or power with respect to *OoO*—in particular *F-32*, *F-64*, and *F-1024* are not considered in this discussion.

Figure 5-19 plots normalized runtime of *OoO*, *F-128*, *F-256*, and *F-512* across single-threaded benchmarks. I find that *F-128*'s performance closely approximates that of *OoO*, and is in fact slightly better on average (about 4%). Chapter 4 demonstrated how an SSR-based core, or a hypothetical out-of-order core using a full-window SSR scheduler, can overcome the serialized wakeup effects by leveraging a larger instruction scheduler. As evident from Figure 5-19, Forwardflow, too, is able to successfully leverage a larger effective scheduler size and improve performance with

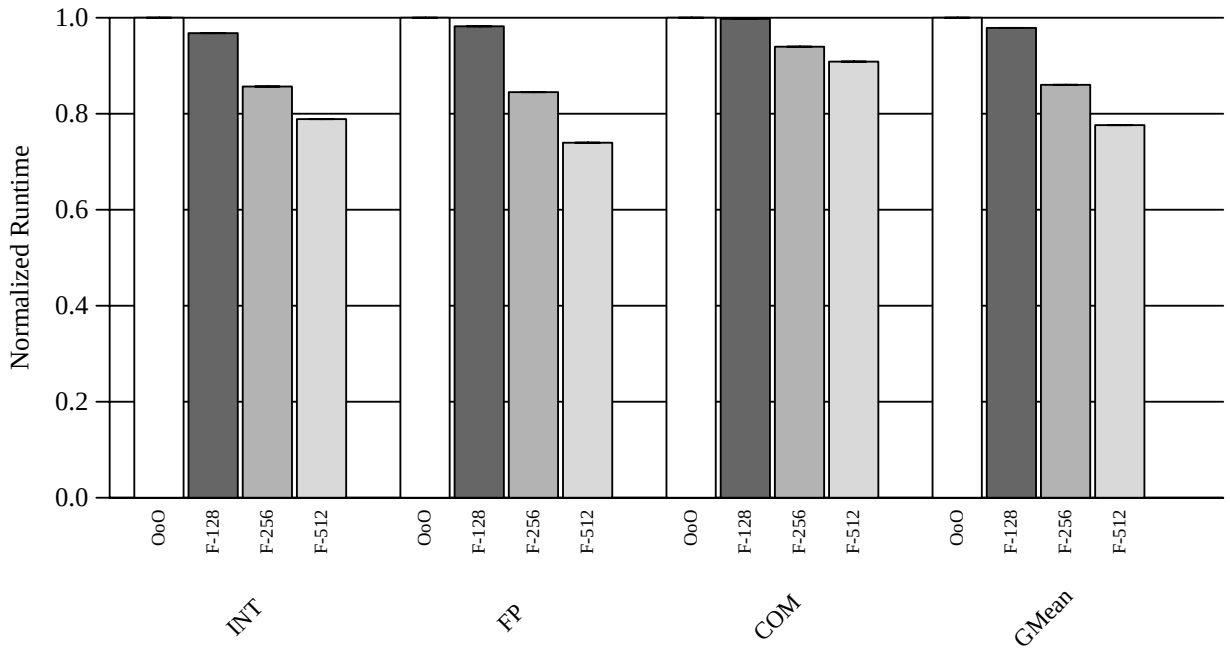


FIGURE 5-19. Normalized runtime of SPEC INT 2006 (INT), SPEC FP 2006 (FP) and Wisconsin Commercial Workloads (COM), *OoO* and Forwardflow configurations with similar power consumption.

respect to *OoO*. This trend is further evidenced by the plot of window occupancy between *OoO* and *F-128* (Figure 5-20—both designs have identically-sized windows, but the latter has a full-window scheduler). With a 32-entry scheduler and a 128-entry ROB, OoO suffers from scheduler clog (cf Chapter 2), yielding poorer overall performance as compared to *F-128*. On average, *F-256* reduces runtime 14% with respect to *OoO*; *F-512* reduces runtime by 22%.

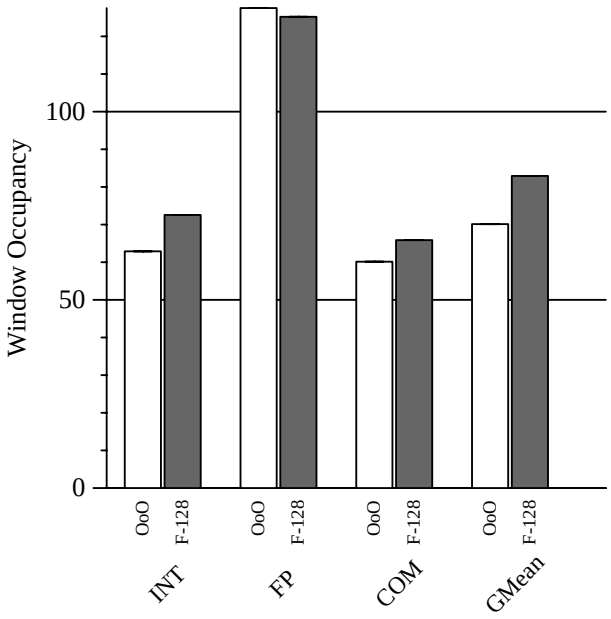


FIGURE 5-20. Window Occupancy of *OoO* and comparable Forwardflow configuration.

Figure 5-21 plots categorized power consumption of each design point. Each group (INT, FP, and COM) is normalized with respect to the average power consumed by *OoO* on that benchmark suite. I categorize power consumption into twelve areas. Of those not self-explanatory, "Sched/UH" encompasses power consumption of the *OoO* scheduler [78] and Forwardflow's update hardware, "Bypass/ON" represents the power consumed in bypass networks and operating networks (the latter is Forwardflow-specific), "DMEM" includes contributions from L1-D caches and D-TLBs, "Network" refers to the on-chip network between caches and the memory controllers, and "Other" includes NoSQ, commit logic, and miscellaneous registers not suitable for other categories. Lastly, "Static/Clk" represent leakage power and contributions from other constant sources, such as the chip's clock signal generators and distribution network. Static power consumption is fairly significant by percentage, as this work assumes only one core of an 8-core CMP operates, but the whole of the large L3 cache is active (details in Chapter 3).

Figure 5-21 yields several interesting findings. First, compared to *OoO*, Forwardflow cores dramatically reduce scheduling, register file, and bypass power. Second, progressively larger Forwardflow cores require more power to operate, but not all additional power is consumed by additional window components: significant increases in power consumption of on-chip caches, register files, and the fetch logic are evident. The physical designs of these elements are unchanged across all

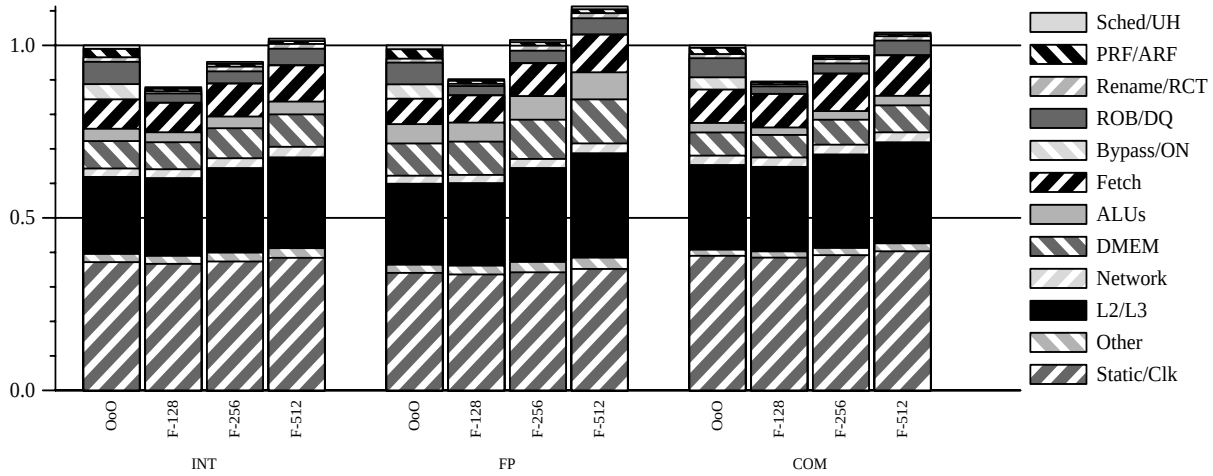


FIGURE 5-21. Categorized power consumption of SPEC INT 2006 (INT), SPEC FP 2006 (FP) and Wisconsin Commercial Workloads (COM), OoO and Forwardflow configurations with similar power consumption.

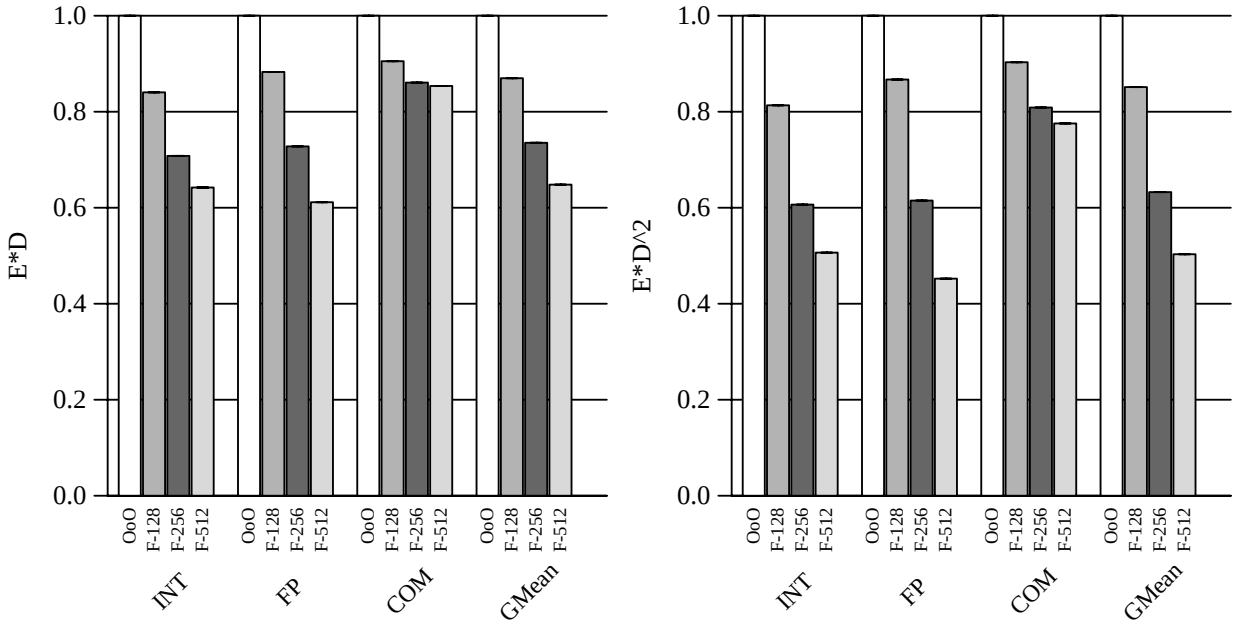


FIGURE 5-22. $E \cdot D$ (left) and $E \cdot D^2$ (right) of SPEC INT 2006 (INT), SPEC FP 2006 (FP), and Wisconsin Commercial Workloads (COM), OoO and Forwardflow configurations with similar power consumption.

Forwardflow configurations in this study: the increase in their power consumption is due to additional demand placed on these structures by larger-window configurations[2].

2. The trends behind Forwardflow's power scaling in relation to window size are more fully discussed in Section 5.4.3.

As a consequence of Forwardflow's reduction in core power requirements, the performance of *F-256* (or even *F-512*) is usually attainable in power comparable to that of *OoO*. This trend varies somewhat by workload—full data is available in Figures 5-37 through 5-40.

Figure 5-22 plots energy efficiency (i.e., $E \cdot D$, left, and $E \cdot D^2$, right) over *OoO* and comparable Forwardflow designs. On average, all Forwardflow configurations are more efficient (by either metric) than *OoO*. In most cases, either the performance gains of larger windows justify the additional power consumption (significant reduction in *D*), or the power savings justify any small performance degradation (significant reduction in *E*). When individual benchmarks are considered, *OoO* is $E \cdot D$- and $E \cdot D^2$-optimal in only three cases: `bzip2`, `gromacs`, and `povray`. Among the remaining 30 benchmarks, *F-128* is most $E \cdot D$-optimal once (`tonto`), and either *F-256* or *F-512* are most optimal for other workloads.

5.4.3 Forwardflow Power Consumption by Window Size

Sections 5.4.1 and 5.4.2 placed the normalized runtime and power estimates in context of previous designs. This section considers how much power/performance range is possible within the Forwardflow design space. To foreshadow subsequent work, this analysis will be important when considering Forwardflow's utility as a scalable core (i.e., Chapter 6).

Figure 5-23 plots normalized runtime across a wide gamut of Forwardflow configurations, ranging from *F-32* to *F-1024*. The two leftmost configurations, *F-32* and *F-64* (hashed in the fig-

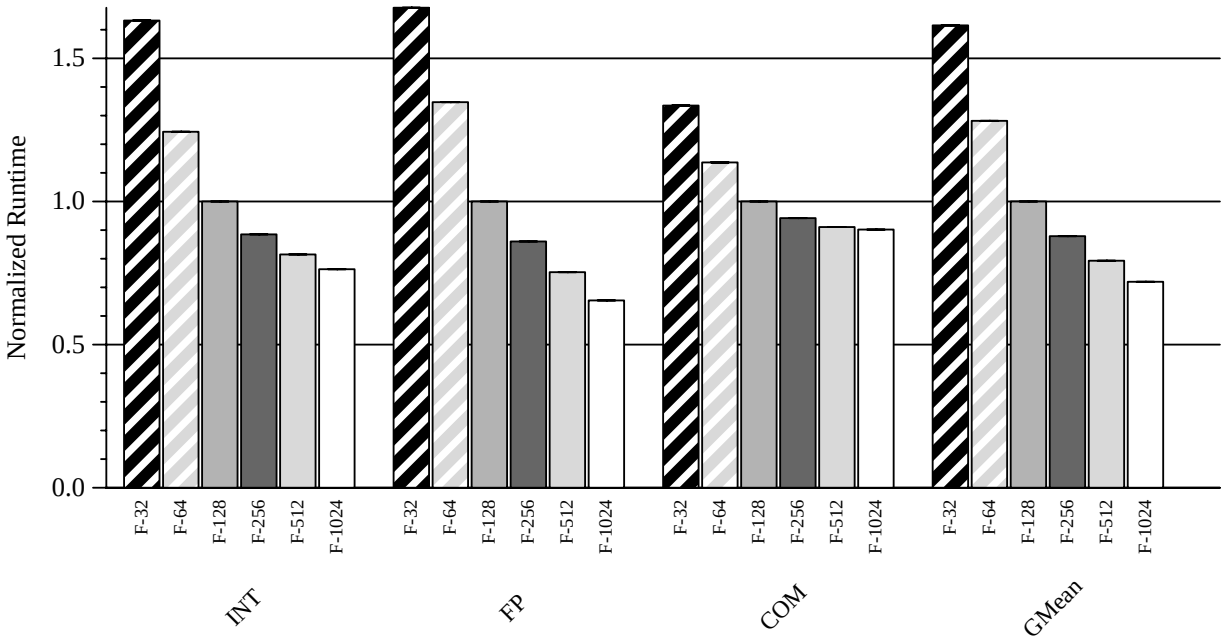


FIGURE 5-23. Normalized runtime of SPEC INT 2006 (INT), SPEC FP 2006 (FP), and Wisconsin Commercial Workloads (COM), Forwardflow designs, 32- through 1024-entry windows.

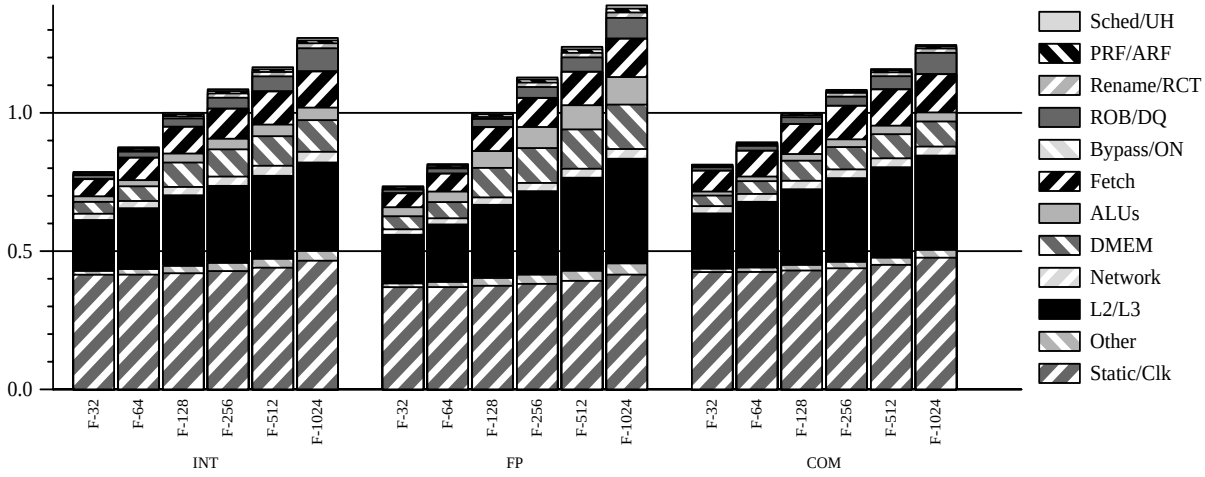


FIGURE 5-24. Categorized power consumption of SPEC INT 2006 (INT), SPEC FP 2006 (FP), and Wisconsin Commercial Workloads (COM), Forwardflow designs, 32- through 1024-entry windows.

ure), implement their small instruction windows by provisioning only one or two DQ banks, respectively, to a singleton DQ bank group. Therefore, these machines have reduced dispatch, issue, and commit width of 1 and 2, respectively (in addition to smaller windows). The adjective

"smaller" in this prose refers, of course, to the instruction window size, but the actual area of these designs are all comparable—recall that each individual DQ bank group occupies an area smaller than 1 mm^2 (Section 5.3.7). Figure 5-24 plots the corresponding power consumption of each design, normalized to *F-128*.

Between *F-32* and *F-1024*, there is a substantial power (e.g., -25% power for *F-32*, +25% power for *F-1024*) and performance (e.g., +65% runtime for *F-32*, -28% runtime for *F-1024*) range available. Scalable CMPs desire scalable cores with a wide power/performance range—Forwardflow fulfills this requirement (Chapter 6).

Figure 5-25 examines Forwardflow's power scaling more closely, by plotting the normalized power consumed by the DQ itself and the fetch logic. Note for this and the subsequent graphs in this section that the x-axis of Figures 5-25 through 5-27 have logarithmic scale.

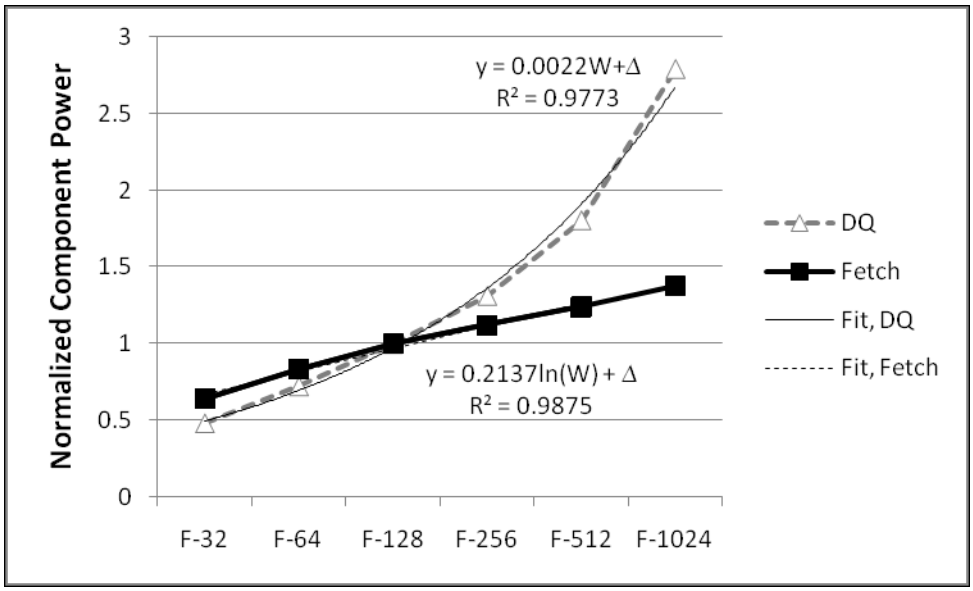


FIGURE 5-25. Normalized component power for DQ and Fetch pipelines, SPEC INT 2006, over six different Forwardflow configurations.

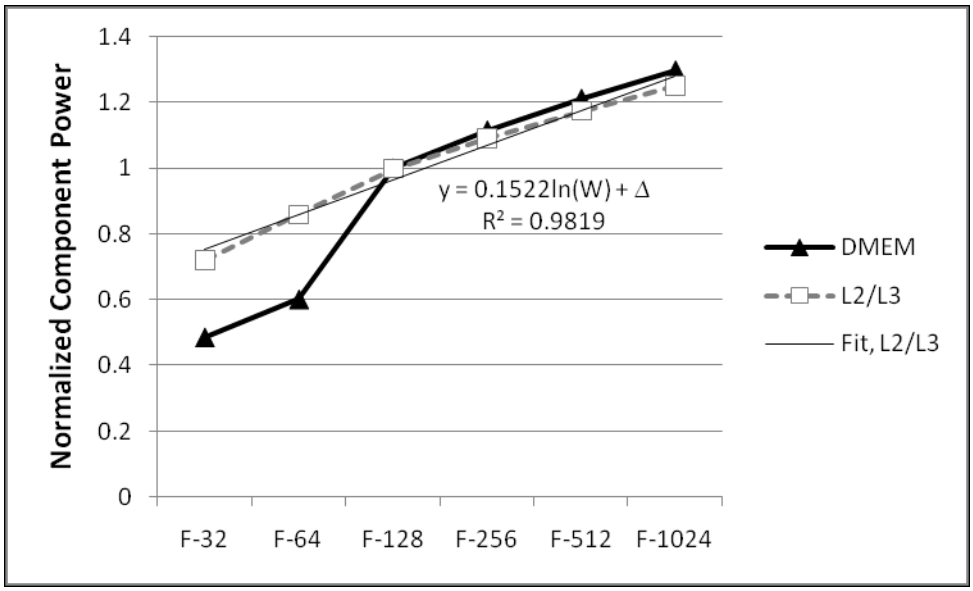


FIGURE 5-26. Normalized component power for memory subsystem, SPEC INT 2006, over six different Forwardflow configurations.

Overall, DQ power grows linearly in window size. This follows from intuition: DQ *area* grows linearly in window size, and assuming a constant activity factor, one would expect power to grow linearly as well. The growth in window size places additional demand on the frontend: in order to maintain a full window, the frontend must provide instructions at greater rates to higher-performing configurations. The fit to the fetch logic in Figure 5-25 indicates that the resulting increase in power consumption is roughly logarithmic in window size.

I observe similar trends in Figure 5-26, which plots self-normalized power consumption of DMEM and L2/L3 components. As with Fetch, demand on the upper-level caches grows roughly logarithmically with window size. DMEM would likely show a similar trend, if *F-32* and *F-64* had maintained the same number of ports on the L1-D cache and D-TLB as *F-128* and larger designs (since the leftmost designs are narrower, fewer ports are provisioned).

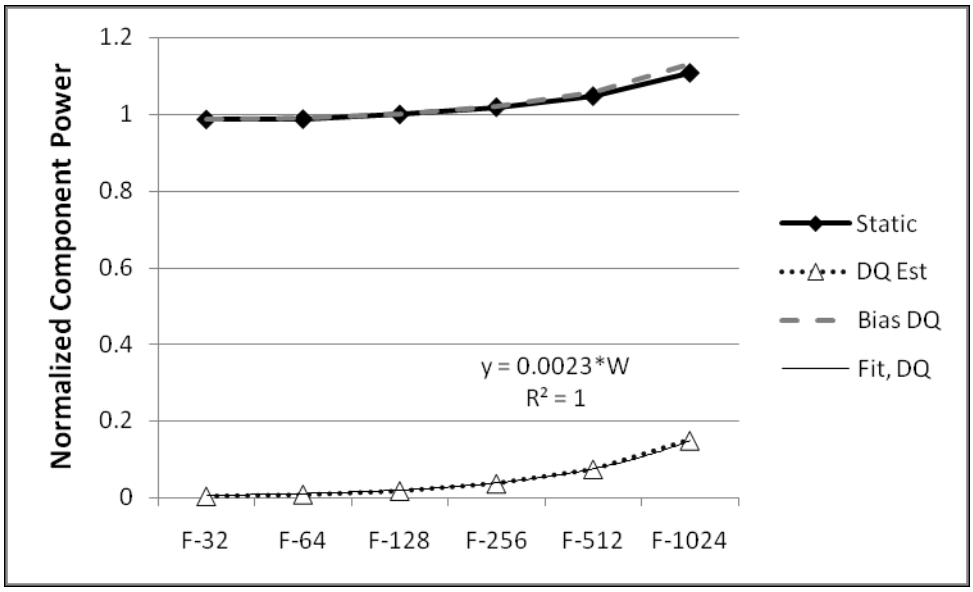


FIGURE 5-27. Normalized component power, static sources, SPEC INT 2006, over six different Forwardflow configurations.

Figure 5-27 considers the growth of static power in progressively larger Forwardflow designs. Additional static power is consumed by the addition of DQ bank groups (recall that each group consists of SRAM arrays, bit vectors, logic, and functional pipelines). There is also a small change in overall chip area as additional bank groups are added to the design, which may affect the power consumed by the clock distribution network (however, recall that DQ bank groups are very small with respect to die area). The estimate of static power consumed by DQ bank groups is plotted on Figure 5-27 as *DQ Est*, which fits very tightly to a linear curve. The line *Bias DQ* is the sum of all other static components (almost unchanged across Forwardflow designs) and *DQ Est*. As evident from the graph, *Bias DQ* closely approximates actual static consumption.

Lastly, overall energy efficiency is plotted in Figure 5-28. Several worthwhile findings present themselves. First, efficiency is not monotonic in window size. E.g., *F-1024* is less efficient (both $E \cdot D$ and $E \cdot D^2$) than smaller designs for commercial benchmarks. These benchmarks derive the

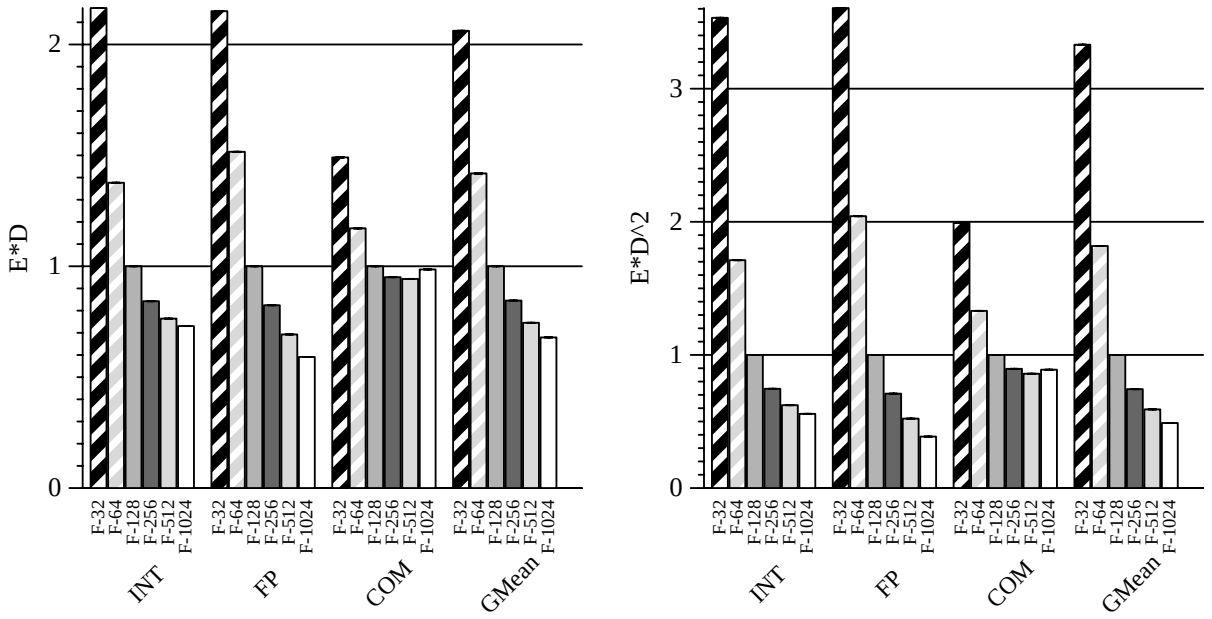


FIGURE 5-28. $E \cdot D$ **(left) and** $E \cdot D^2$ **(right) of SPEC INT 2006 (INT), SPEC FP 2006 (FP), and Wisconsin Commercial Workloads (COM), over six different Forwardflow configurations.**

least benefit from larger windows, but still suffer power overheads commensurate with window size.

Second, neither *F-32* or *F-64* are ever optimal (either $E \cdot D$ or $E \cdot D^2$) over the intervals I have examined. Complete data (Section 5.4.7) shows that, for individual workloads, all other Forwardflow sizes are occasionally optimal. Of course, it is possible to construct workloads for which the smallest configurations are optimal, but for these workloads the great reduction in performance observed by reducing the effective dispatch/commit width does not justify the modest power reduction of *F-32* and *F-64*.

5.4.4 Forwardflow Comparison to Runahead and CFP

I next evaluate the Forwardflow design point against two prior MLP-aware microarchitectural enhancements: *Runahead Execution* [46, 122], typified by the *Runahead* model, and *Continual*

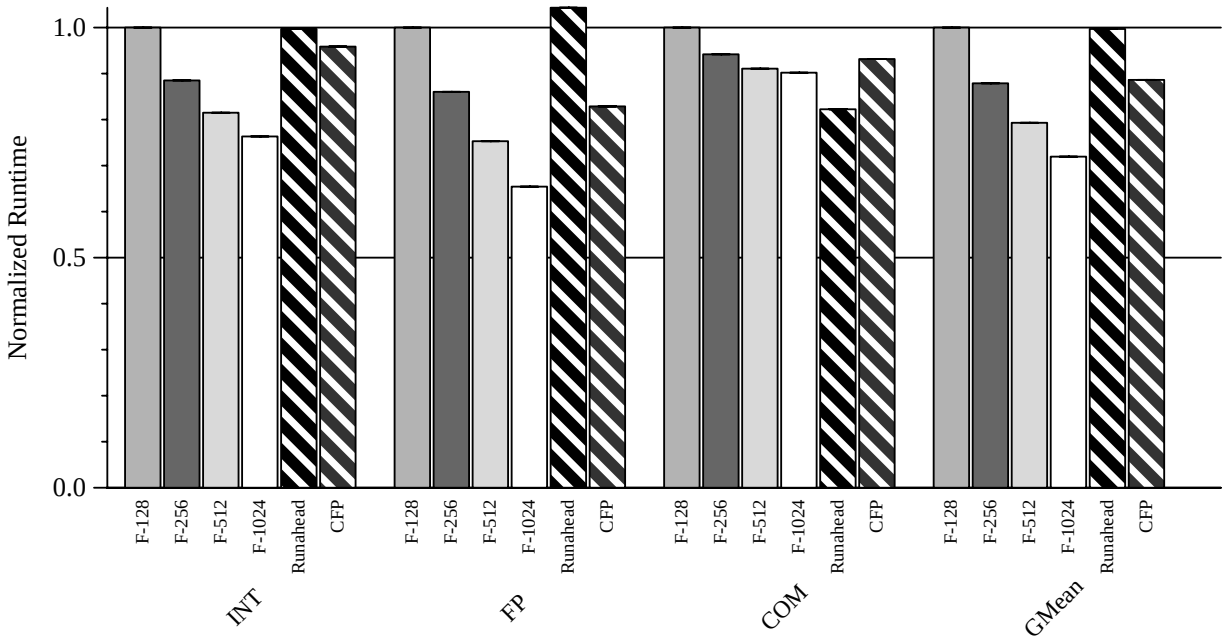


FIGURE 5-29. Normalized runtime of SPEC INT 2006 (INT), SPEC FP 2006 (FP), and Wisconsin Commercial Workloads (COM), Forwardflow designs, *Runahead*, and *CFP*.

Flow Pipelines [156], typified by *CFP*. Figure 5-29 plots normalized runtime of full-width Forwardflow configurations, compared to that of *Runahead* and *CFP* models (details of these models are in Chapter 3). Overall, performance of *Runahead* is not far from that reported by Mutlu et al. for a similar benchmark suite (the majority of results in that work show better performance for benchmarks with higher overall IPC than those used in this study). Looking ahead to Figures 5-31 through 5-35, *Runahead* tends to outperform *OoO*, as expected (*OoO* does not appear on Figure 5-29). Observed results for *CFP* are slightly better than those reported by Srinivasan et al., likely due to some degree of idealization in the *CFP* model.

Overall, *Runahead* is not able to issue enough useful prefetches in the integer and floating point codes to make up for the performance loss due to end-of-runahead pipeline squashes. In the case of the latter, the opportunity loss of delaying throughput-critical floating point operations yields a slight performance loss, compared to *F-128*. As indicated by Figure 5-17, even large-win-

dow *RUU* configurations do not expose as much MLP in the integer suite as in SPEC FP. However, Runahead performs well for two of the four commercial workloads (`apache` and `zeus`). Both of these workloads exhibit more complicated data dependences than does the SPEC suite, and fairly low IPC overall. However, in runahead mode, these dependences quickly evaporate as *Runahead* discards dependant instructions. As a result, *Runahead* is able to achieve a runahead-mode IPC[3] much higher than that implied by the dataflow limits of the benchmark themselves. *Runahead* effectively discards instructions that tend to limit IPC in these workloads, e.g., faulting instructions and improperly-disambiguated loads. Furthermore, the *Runahead* implementation can totally discard serializing events, such as speculatively-inlined exceptions. Though these events usually do not trigger a pipeline flush, they cannot be safely executed until they reach the eldest position in the DQ (or ROB, depending on the microarchitectural model).

These two effects enable the *Runahead* model to issue memory requests thousands of instructions apart—whereas microarchitectures that do not discard dependent instructions are eventually limited by squashes, and maintain only about 190 instructions in-flight on average, even with thousand-entry windows (cf Figure 5-18). In other words, `apache` and `zeus` do exhibit independence, but the limitations of control prediction and serialization can inhibit some microarchitectures' ability to discover independent instructions.

CFP, on the other hand, improves performance by making better use of a small scheduler (through instruction deferral and re-dispatch). Without end-of-runahead squashes, *CFP* outperforms *Runahead* in most cases, reaching a performance level roughly comparable to *F-256*.

3. Runahead-mode IPC is the rate at which instructions are discarded during runahead mode. It is distinct from IPC, which is the rate at which instructions commit. IPC is zero during Runahead mode.

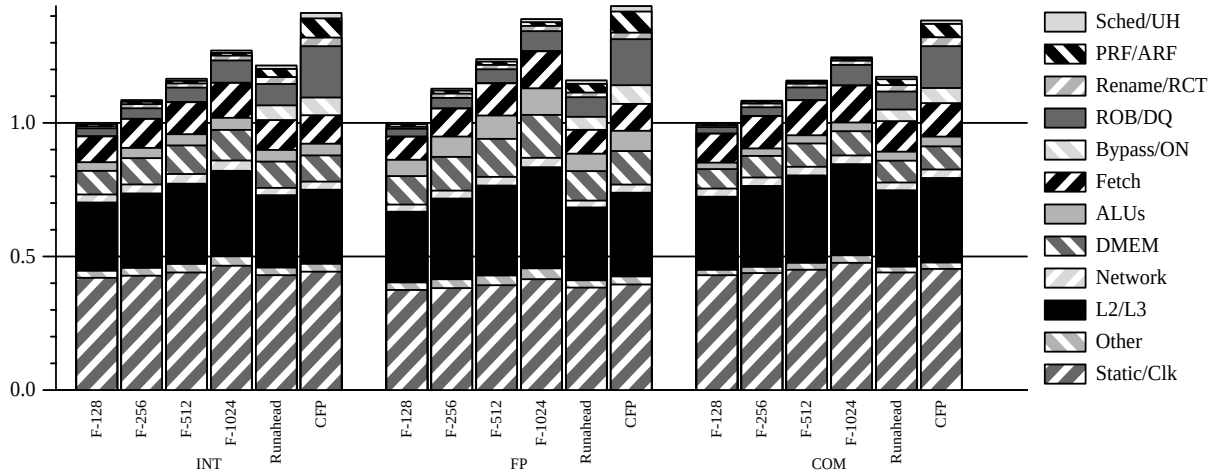


FIGURE 5-30. Categorized power of SPEC INT 2006 (INT), SPEC FP 2006 (FP), and Wisconsin Commercial Workloads (COM), Forwardflow designs, *Runahead*, and *CFP*.

Figure 5-30 plots the power consumption of *Runahead* and *CFP* in the context of comparable Forwardflow design points. Power consumed by *CFP*'s deferred queues are added to the "ROB/DQ" category (note: "DQ" in this context abbreviates "Dataflow Queue," referring to the instruction window in Forwardflow). *Runahead*'s runahead cache is added to the consumption of the "DMEM" category, as it is accessed in parallel to the L1-D.

Overall, *Runahead* and *CFP* operate at power consumption comparable to or greater than the largest Forwardflow configurations. In addition the inefficiencies of the underlying *OoO* design, *Runahead* tends to increase the overall activity factor of the core, by eliminating memory stall periods. Similarly, *CFP*'s DQs are in near-constant operation, deferring up to a thousand instructions at a time in search of ILP. Unlike a Forwardflow DQ, which has predictable access patterns for dispatch and commit, *CFP*'s DQs access patterns are more random, as slice resolution opportunistically defers, and potentially, multiply *re-defers*, instructions. As such, *CFP*'s DQs are harder to decompose into independent banks than Forwardflow's DQ. *CFP* exhibits substantially higher

"ROB/DQ" power than other designs, due to the dual effect of higher activity factor (e.g., from multiple deferral) and from implementation using larger overall SRAMs.

Interestingly, *Runahead*'s power does not reach that of *F-1024* on the floating point benchmarks, though it does significantly exceed the power of *OoO* (nearly the same as *F-256*). This is a result of power-saving techniques applied in the *Runahead* model, suggested by Mutlu et al. [121]. *Runahead* discards instructions not likely to be related to address calculation (e.g., floating point operations), and attempts to identify short runahead periods and simply stall the pipeline instead of entering runahead mode. These effects are observable in "ALU" and "DMEM" power consumption more comparable to *F-128* than *F-1024*.

5.4.5 Outlier Discussion

In this final section of the quantitative evaluation of Forwardflow, I address atypical Forwardflow behaviors. Much of the data explained in this section refers to the unabridged graphs in Section 5.4.7, following this chapter's conclusion. This evaluation's main points are easier to follow if these graphs are deferred until the end of the chapter.

Excellent performance outliers: `libquantum`, `GemsFDTD`, `milc`, and to a lesser extent, `bwaves`. `libquantum` and benchmarks like it are extremely sensitive to lookahead, and fairly computationally sparse. In other words, they are near-ideal benchmarks for microarchitectures optimizing MLP, like Forwardflow. *CFP* also tends to perform well on these benchmarks. Each of these benchmarks consists of a simple inner loop, with loop-invariant address calculations. A strided prefetcher would likely be very effective for these workloads, though I have not quantified this potential.

Poor performance outliers: `bzip2`, `povray`, `hmmer`, and, to a lesser extent, `gromacs`. Just as the inherent characteristics of some benchmarks are well-suited to Forwardflow, others are less so. Chapter 4 grants some insight into this pathological behavior: `bzip2` and `hmmer`, for instance, suffer from performance loss due to serialized wakeup, inherent in an SSR core. Still other benchmarks (e.g., `povray`, and some listed below) saturate a particular resource in Forwardflow, such as ARF write bandwidth.

Non-scalable benchmarks: `gcc`, `gobmk`, `omnetpp`. Some workloads are simply insensitive to lookahead, as they exhibit relatively little instruction- or memory-level parallelism. `omnetpp` is a prime example of this behavior: much of the portion of this benchmark used in this study is a traversal of a large linked list. A dependence exists between loop iterations, and very little independent work can be discovered. Similarly, `gcc` performs optimizations on an abstract syntax tree, with similar graph-traversal limitations. Still other benchmarks (e.g., commercial workloads) exhibit control flow patterns too difficult even for the aggressive branch predictor used in this study—the expected number of true-path in-flight instructions is low compared to the window size, and false-path instructions do not often issue useful prefetches.

Low power outliers: `libquantum`, `mcf`, `lbm`, `milc`. At first glance at Figures 5-37 through 5-40, some of these outliers seem more like "High power" outliers. This is an artifact of per-benchmark normalization. This evaluation assumes future chip generations will employ fairly aggressive clock gating of unused components. In other words, benchmarks exhibiting long stalls will consume relatively little (dynamic) power during these events. This effect is evidenced on individually-normalized power graphs (e.g., Figure 5-37, benchmark `libquantum`) as a high static-to-

dynamic power ratio. All models assume this same assumption, resulting in lower power compared to that of other benchmarks.

5.4.6 Evaluation Conclusion

I conclude this evaluation with a brief summary of key findings relating to the performance and power characteristics of Forwardflow:

- Forwardflow derives improved performance as window size scales up. Much of this benefit varies with exposed MLP (e.g., median $R^2 = 0.93$).
- Overall, Forwardflow's performance is comparable to that of a similarly-sized traditional out-of-order core. However, Forwardflow's window scales gracefully to larger sizes, through simple replication of DQ bank groups.
- The Forwardflow design is highly energy-efficient, by $E \cdot D$ and $E \cdot D^2$ metrics. However, reducing effective dispatch/commit width is seldom efficient.
- As the Forwardflow window scales, window power increases linearly, but energy demand from other resources scales as well (approximately logarithmically in window size), demonstrating that scaling window size can scale overall chip power consumption.
- Forwardflow's scalable window approach to MLP exploration often allows it to exceed the performance of Runahead Execution and Continual Flow Pipelines.

These key findings will be used in Chapter 6, as I consider the utility of Forwardflow as a dynamically scalable core.

5.4.7 Unabridged Data

This final section of Chapter 5 presents complete data (i.e., runtime, power, efficiency) from all benchmarks in SPEC CPU 2006 and the Wisconsin Commercial Workload Suite. Note that tabular data can be found in Appendix C.

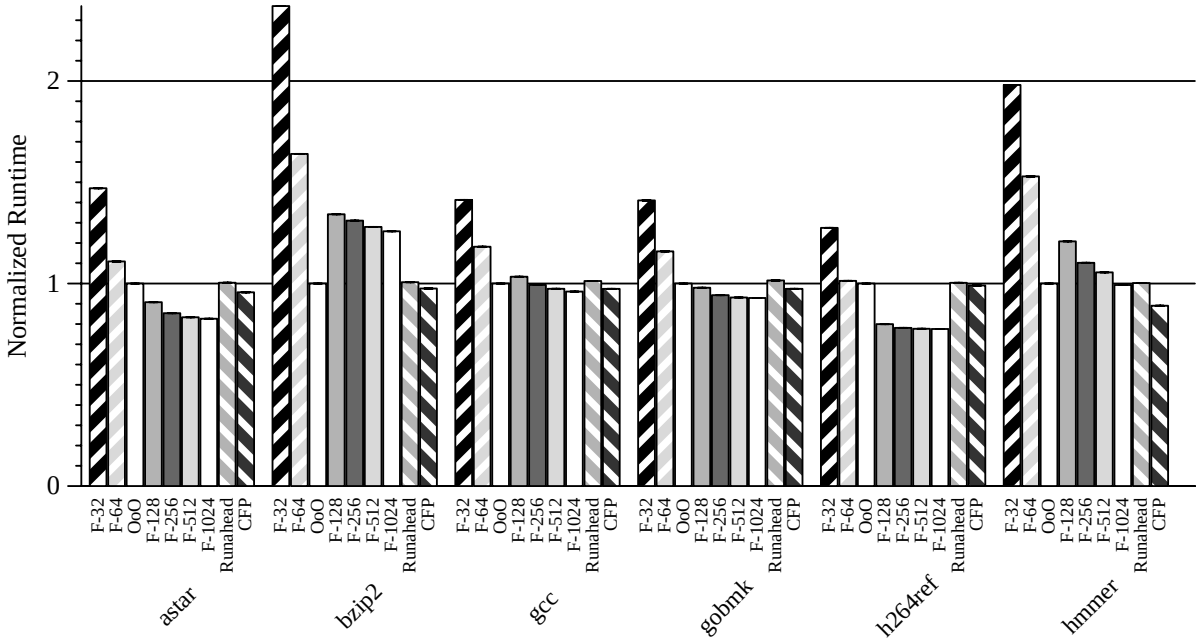


FIGURE 5-31. Normalized runtime, benchmarks `astar`, `bzip2`, `gcc`, `gobmk`, `h264ref`, and `hmmer` (from SPEC INT 2006), all designs.

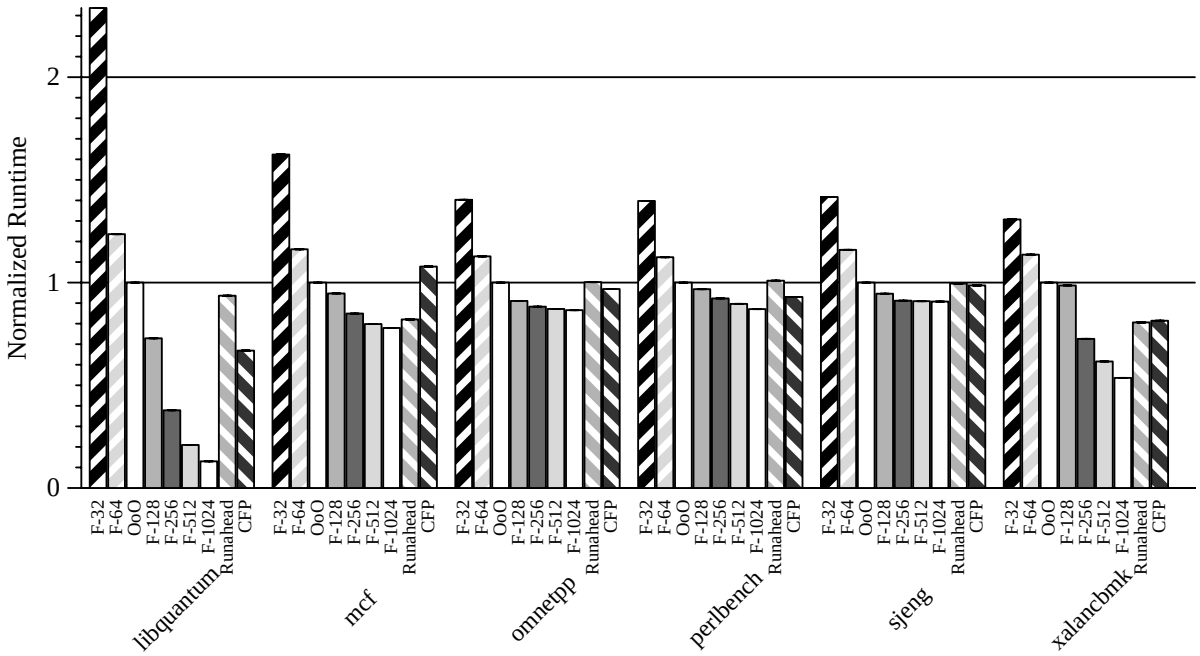


FIGURE 5-32. Normalized runtime, benchmarks `libquantum`, `mcf`, `omnetpp`, `perlbench`, `sjeng`, and `xalancbmk` (from SPEC INT 2006), all designs.

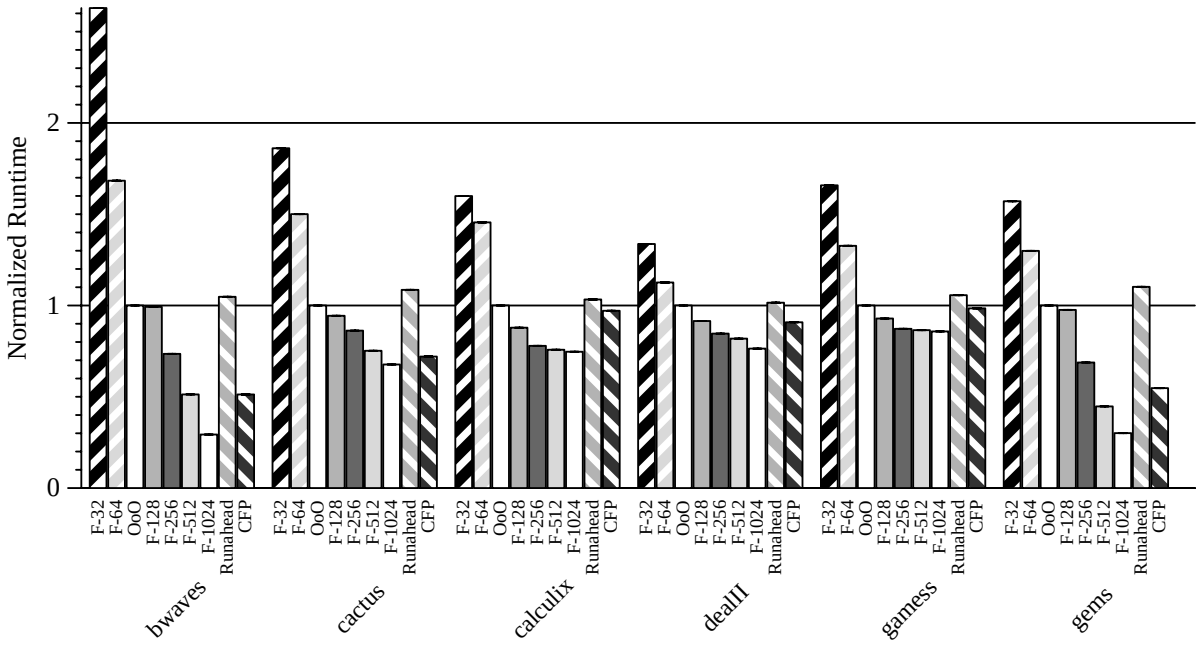


FIGURE 5-33. Normalized runtime, benchmarks `bwaves`, `cactusADM`, `calculix`, `dealII`, `gamess`, and `GemsFDTD` (from SPEC FP 2006), all designs.

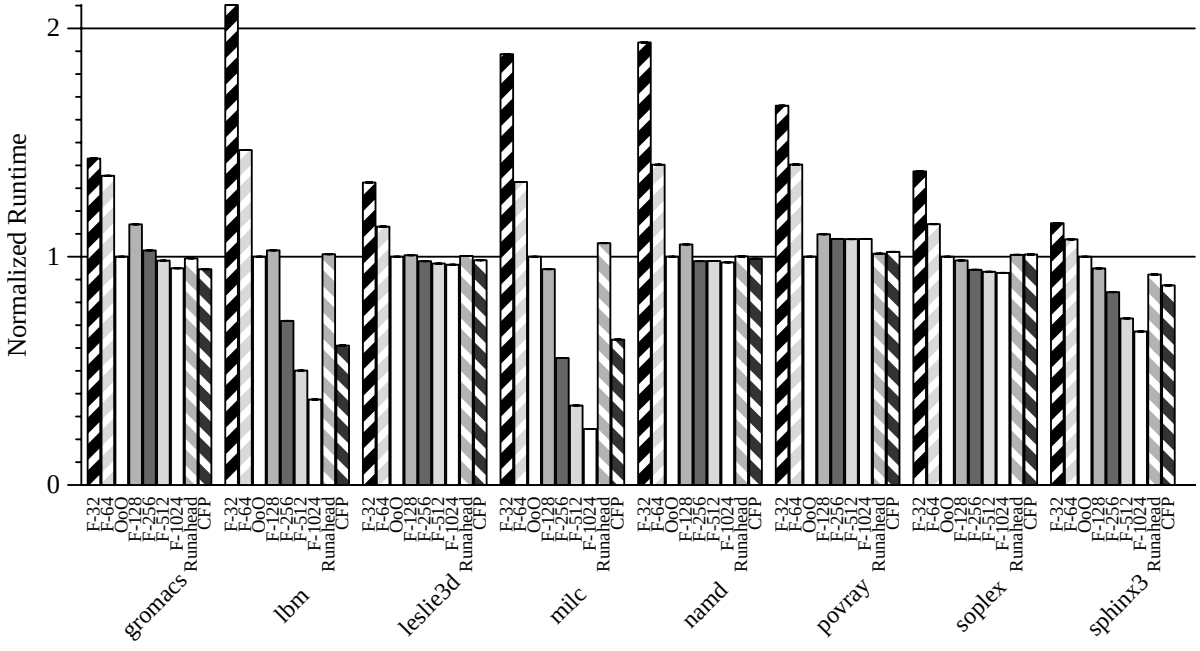


FIGURE 5-34. Normalized runtime, benchmarks `gromacs`, `lbm`, `leslie3d`, `milc`, `namd`, `povray`, `soplex`, and `sphinx3` (from SPEC FP 2006), all designs.

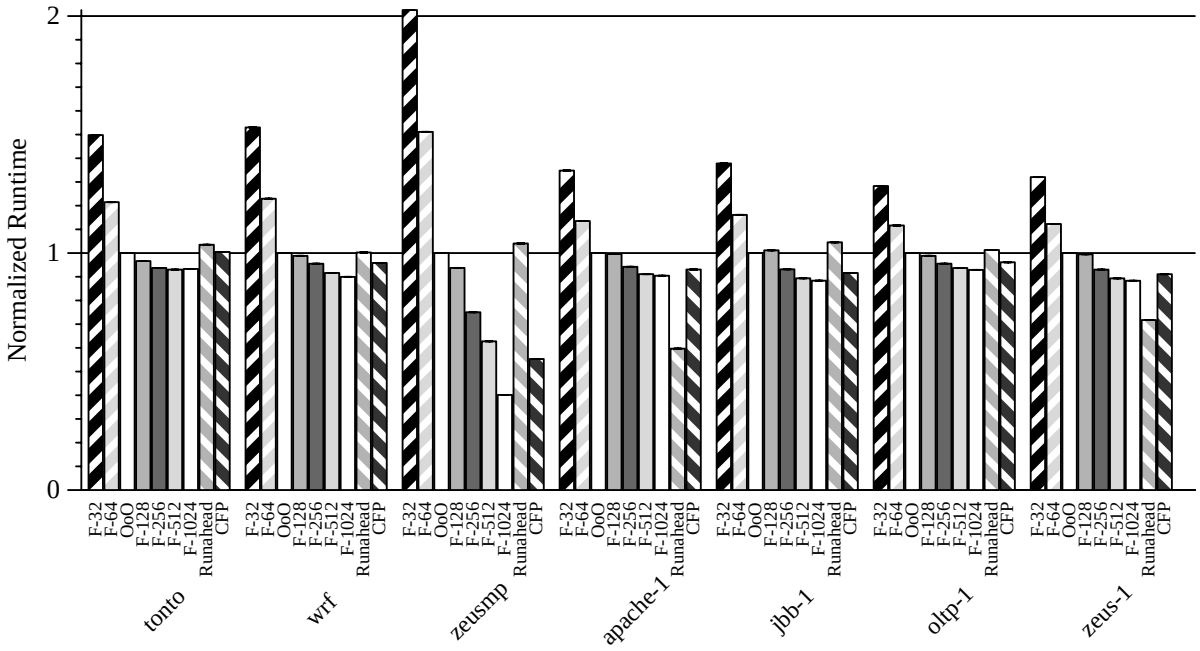


FIGURE 5-35. Normalized runtime, benchmarks `tonto`, `wrf`, `zeusmp` (from SPEC FP 2006), `apache`, `jbb`, `oltp`, and `zeus` (from Wisconsin Commercial Workloads), all designs.

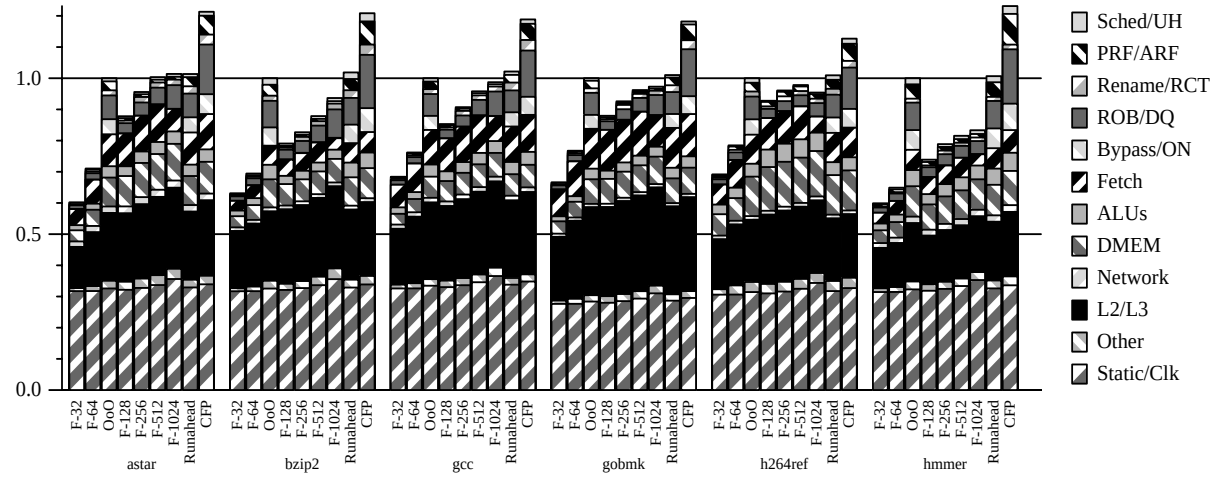


FIGURE 5-36. Categorized power, benchmarks `astar`, `bzip2`, `gcc`, `gobmk`, `h264ref`, and `hmmer` (from SPEC INT 2006), all designs.

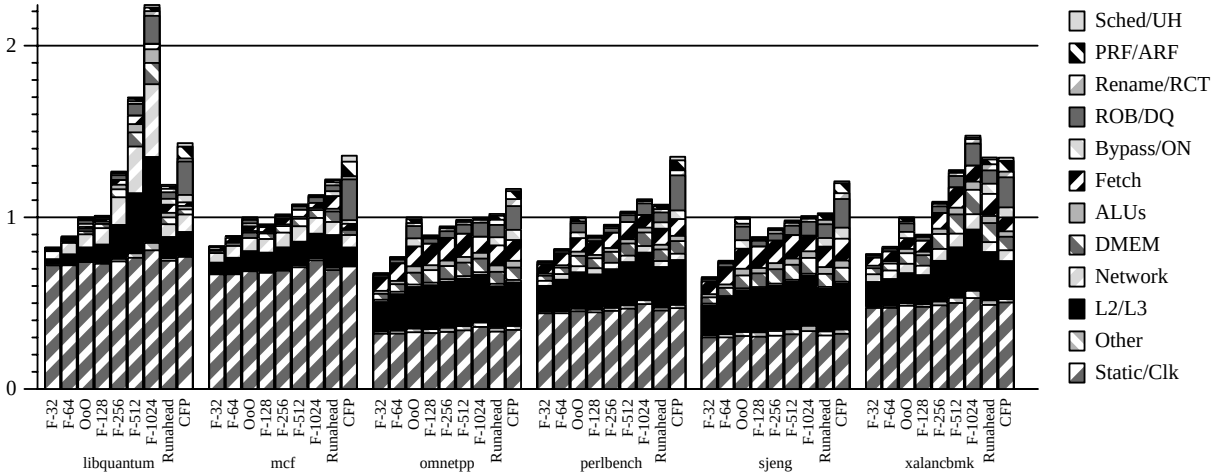


FIGURE 5-37. Categorized power, benchmarks `libquantum`, `mcf`, `omnetpp`, `perlbench`, `sjeng`, and `xalancbmk` (from SPEC INT 2006), all designs.

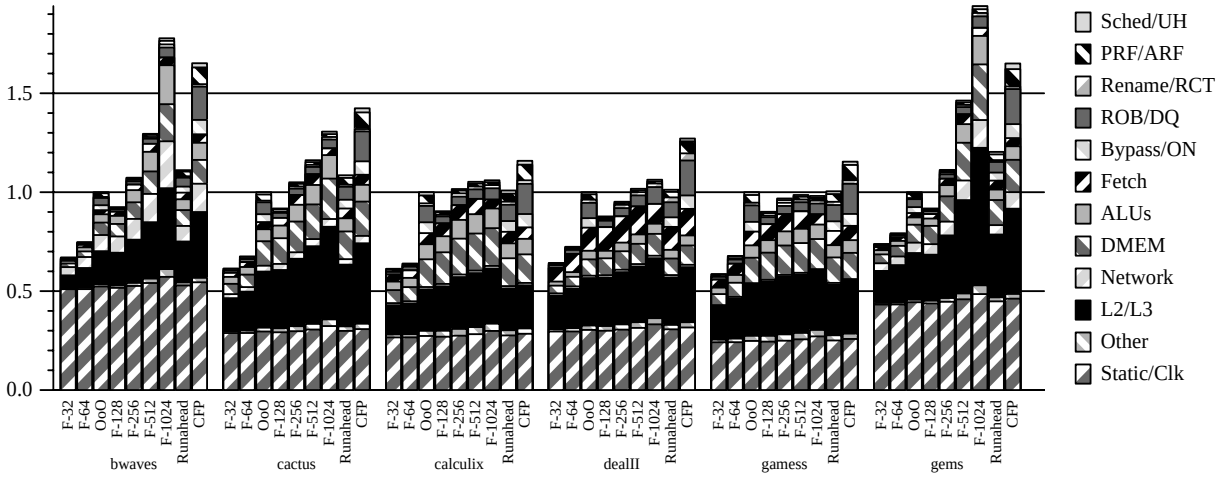


FIGURE 5-38. Categorized power, benchmarks `bwaves`, `cactusADM`, `calculix`, `dealII`, `gamess`, and `GemsFDTD` (from SPEC FP 2006), all designs.

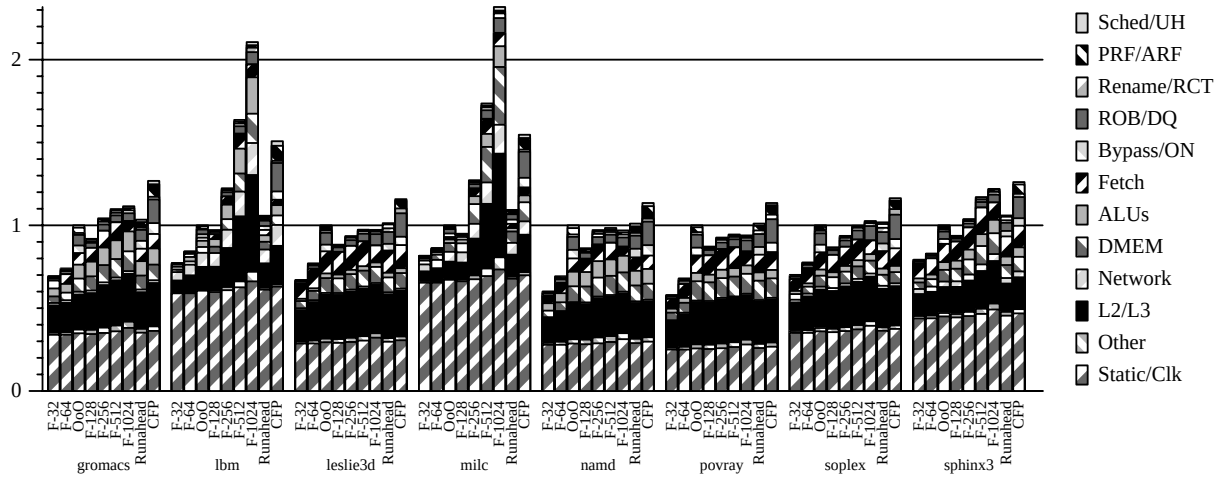


FIGURE 5-39. Categorized power, benchmarks `gromacs`, `lbm`, `leslie3d`, `milc`, `namd`, `povray`, `soplex`, and `sphinx3` (from SPEC FP 2006), all designs.

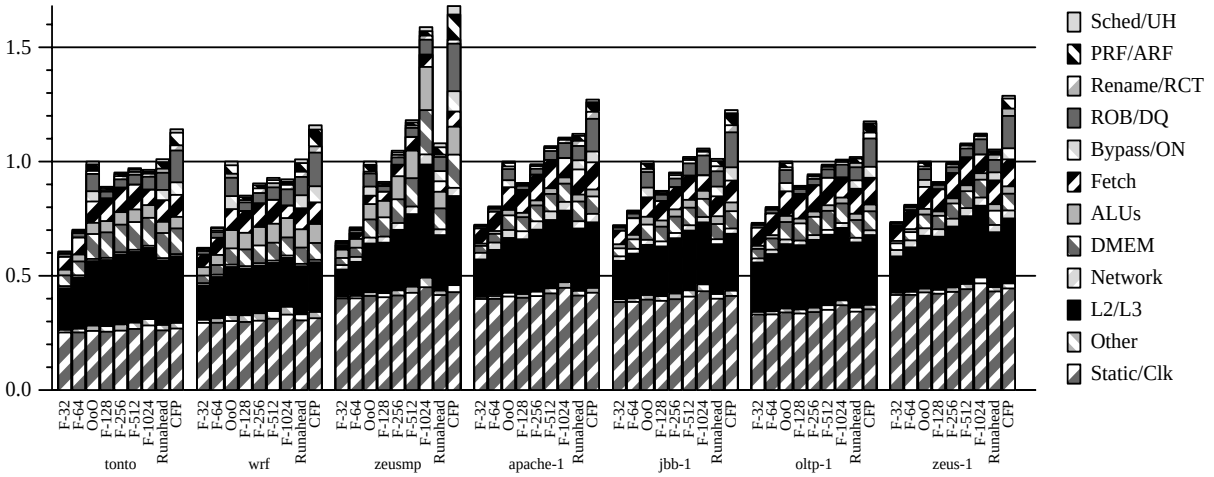


FIGURE 5-40. Categorized power, benchmarks `tonto`, `wrf`, `zeusmp` (from SPEC FP 2006), `apache`, `jbb`, `oltp`, and `zeus` (from Wisconsin Commercial Workloads), all designs.

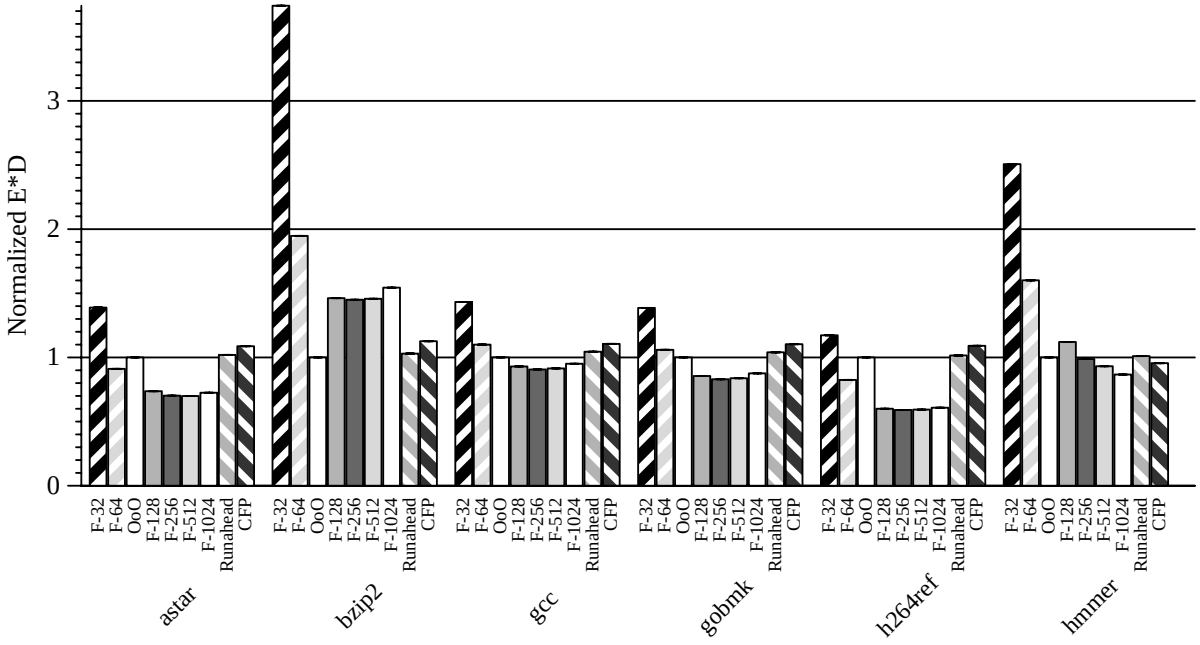


FIGURE 5-41. Normalized $E \cdot D$, benchmarks `astar`, `bzip2`, `gcc`, `gobmk`, `h264ref`, and `hmmer` (from SPEC INT 2006), all designs.

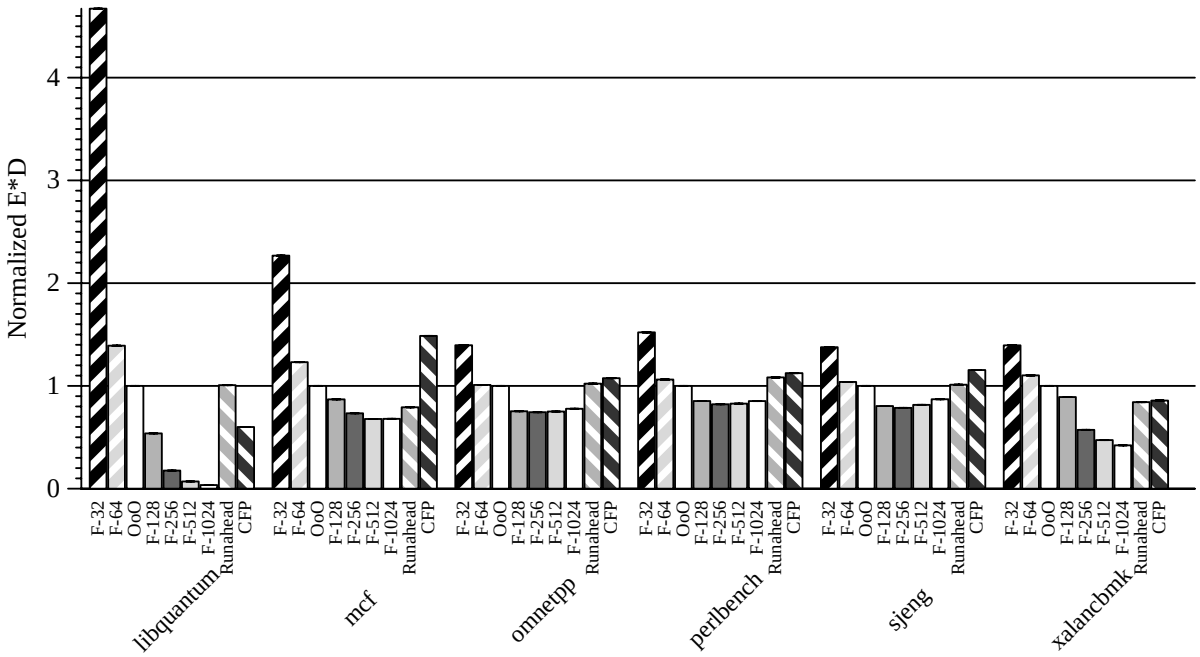


FIGURE 5-42. Normalized $E \cdot D$, benchmarks `libquantum`, `mcf`, `omnetpp`, `perlbench`, `sjeng`, and `xalancbmk` (from SPEC INT 2006), all designs.

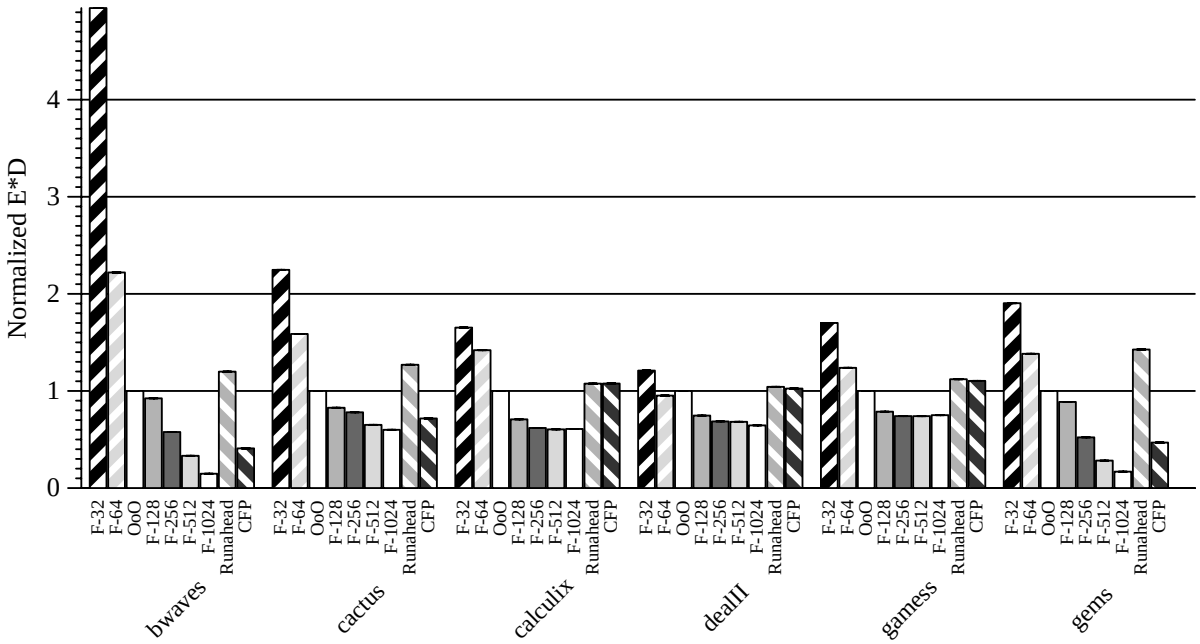


FIGURE 5-43. Normalized $E \cdot D$, benchmarks `bwaves`, `cactusADM`, `calculix`, `dealII`, `gamess`, and `GemsFDTD` (from SPEC FP 2006), all designs.

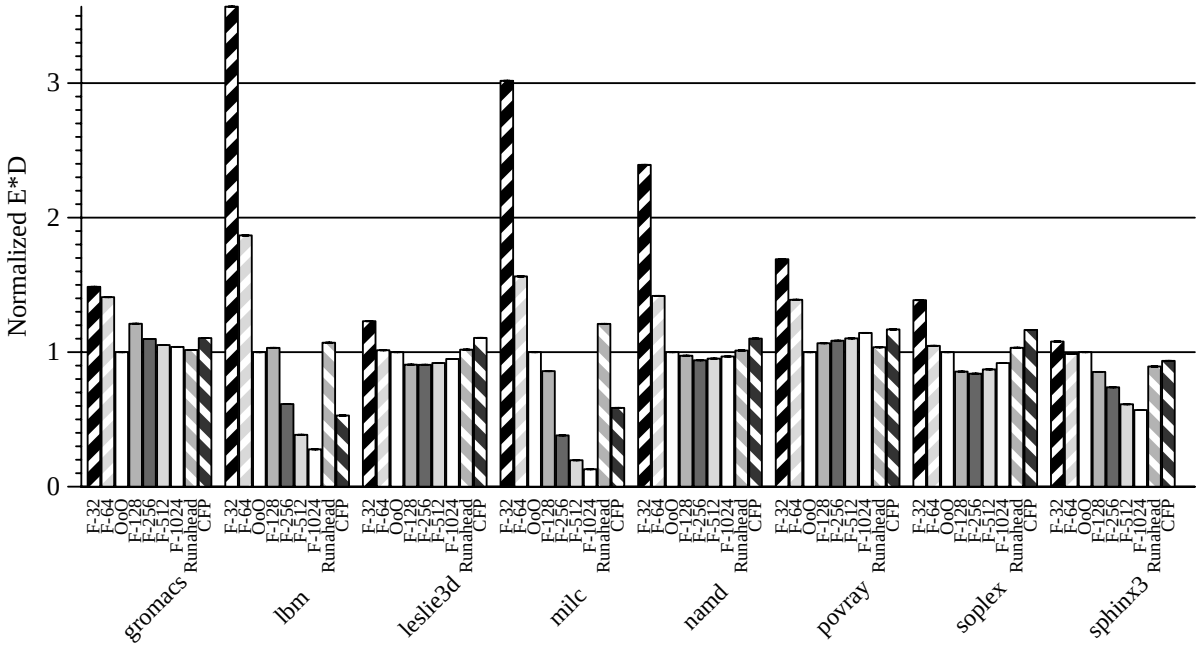


FIGURE 5-44. Normalized $E \cdot D$, benchmarks `gromacs`, `lbm`, `leslie3d`, `milc`, `namd`, `povray`, `soplex`, and `sphinx3` (from SPEC FP 2006), all designs.

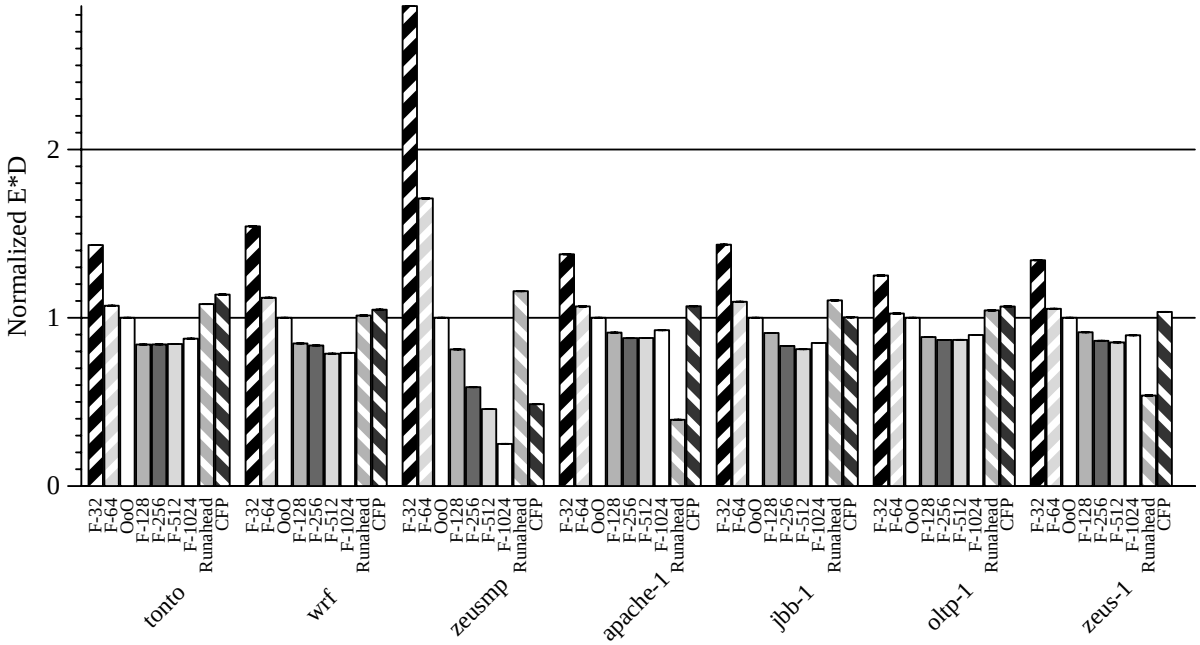


FIGURE 5-45. Normalized $E \cdot D$, benchmarks `tonto`, `wrf`, `zeusmp` (from SPEC FP 2006), `apache`, `jbb`, `oltp`, and `zeus` (from Wisconsin Commercial Workloads), all designs.

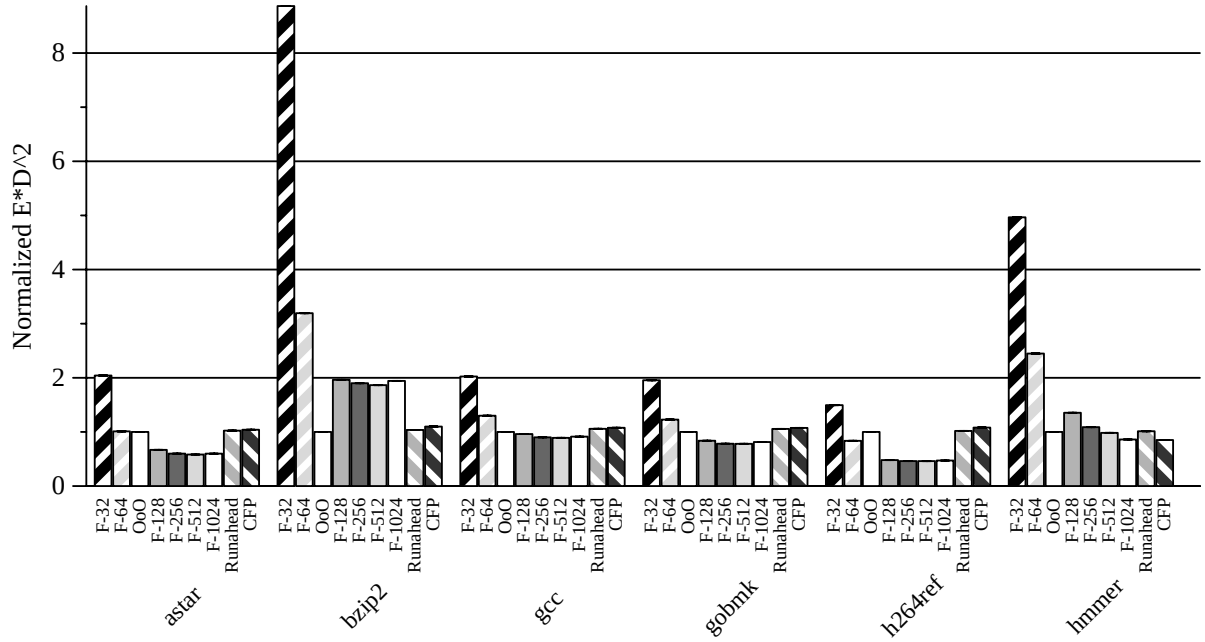


FIGURE 5-46. Normalized $E \cdot D^2$, benchmarks `libquantum`, `mcf`, `omnetpp`, `perlbench`, `sjeng`, and `xalancbmk` (from SPEC INT 2006), all designs.

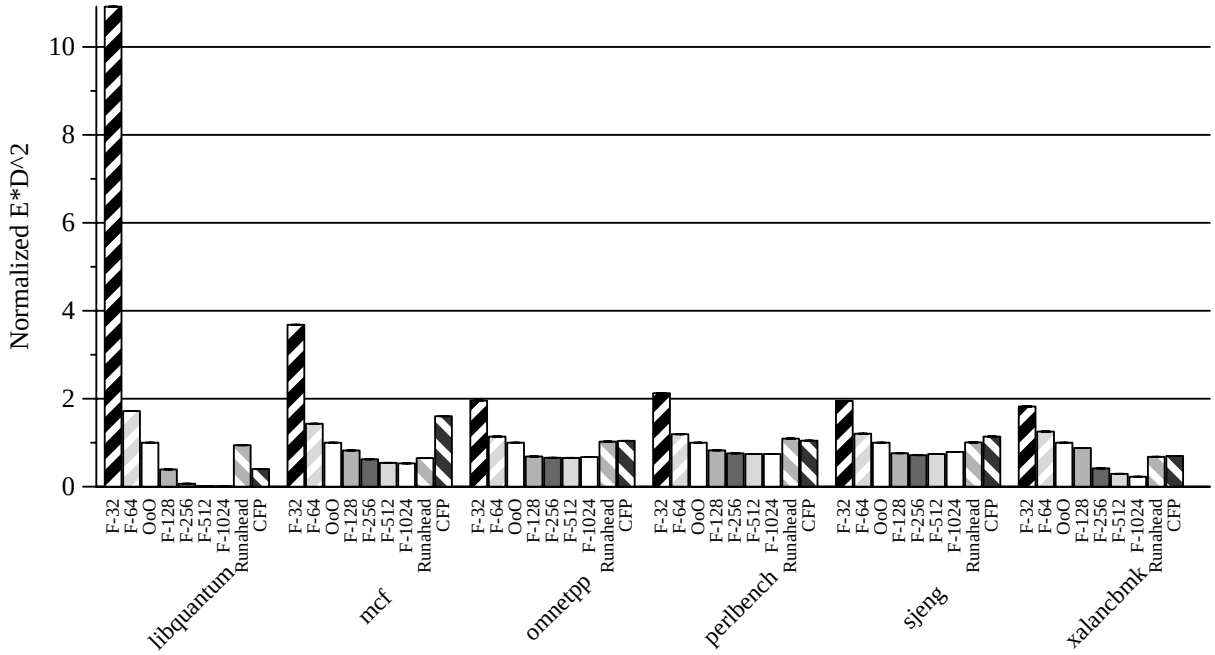


FIGURE 5-47. Normalized $E \cdot D^2$, benchmarks `astar`, `bzip2`, `gcc`, `gobmk`, `h264ref`, and `hmmer` (from SPEC INT 2006), all designs.

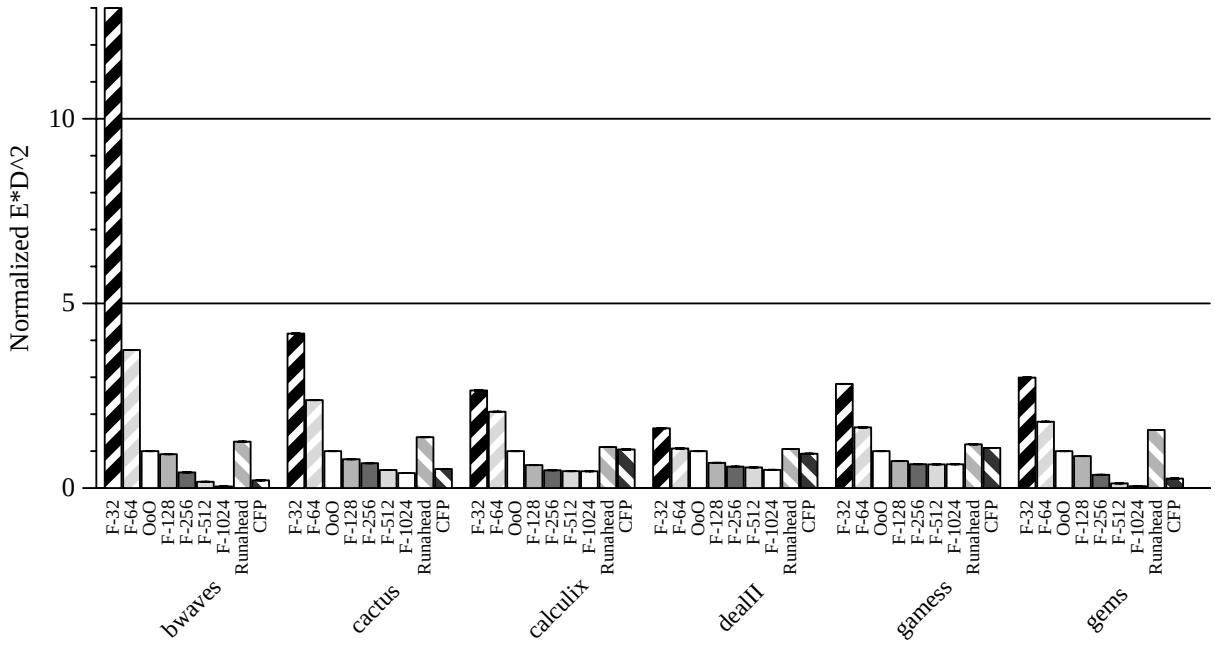


FIGURE 5-48. Normalized $E \cdot D^2$, benchmarks `bwaves`, `cactusADM`, `calculix`, `dealII`, `gamess`, and `GemsFDTD` (from SPEC FP 2006), all designs.

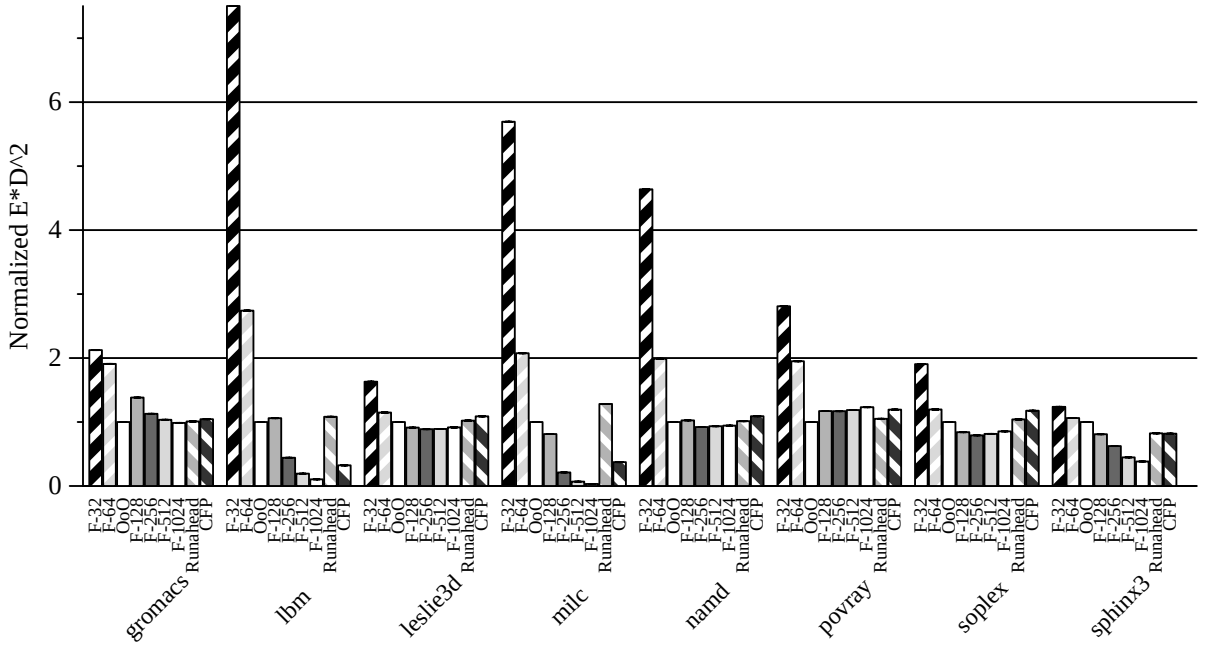


FIGURE 5-49. Normalized $E \cdot D^2$, benchmarks `gromacs`, `lbm`, `leslie3d`, `milc`, `namd`, `povray`, `soplex`, and `sphinx3` (from SPEC FP 2006), all designs.

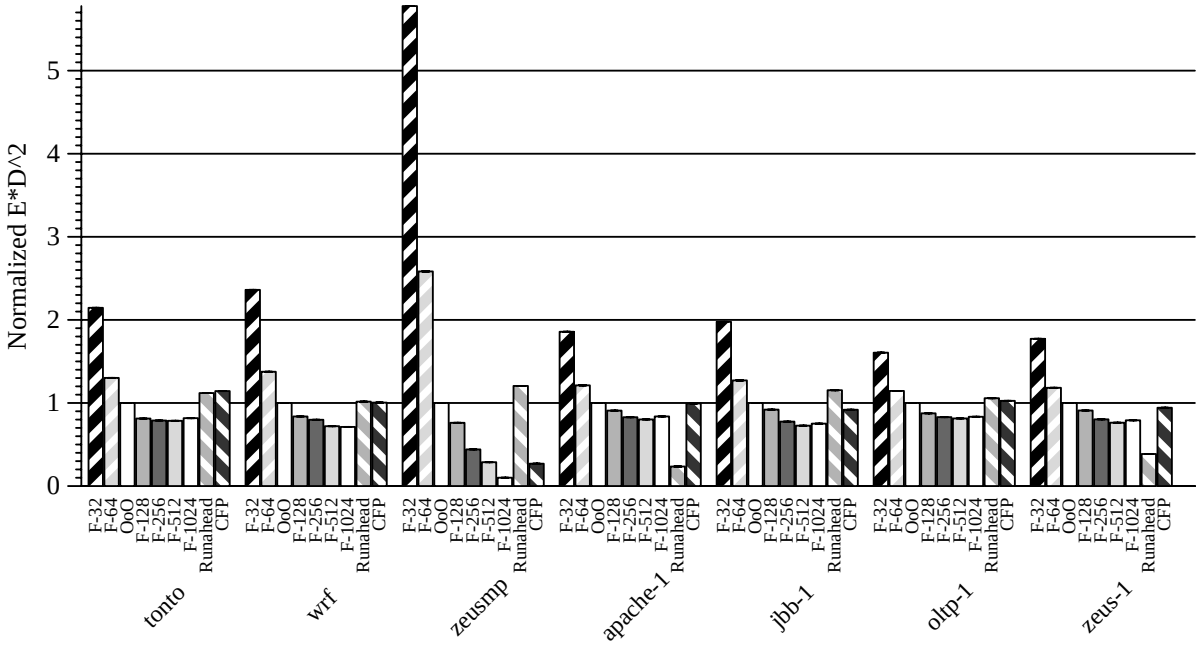


FIGURE 5-50. Normalized $E \cdot D^2$, benchmarks `tonto`, `wrf`, `zeusmp` (from SPEC FP 2006), `apache`, `jbb`, `oltp`, and `zeus` (from Wisconsin Commercial Workloads), all designs.

Chapter 6

Scalable Cores in CMPs

Chapters 4 and 5 developed the foundations for a statically-scalable core, Forwardflow. Chapter 6 discusses how to use Forwardflow cores effectively in a scalable chip multiprocessor (CMP). I begin this discussion with a review of the tradeoffs of scalable CMPs. In particular, the workloads of future chips will be varied: new highly-concurrent codes will emerge, and today's mostly-sequential codes will likely endure as well. Between these two extremes, there will be a variety of partially-parallel executions, hobbled by occasional sequential bottlenecks. A scalable CMP seeks to address *all* of these workloads.

First, when running highly-threaded applications, all cores should *scale down*, each reducing its power/performance point, to afford the power necessary to operate all cores simultaneously. At the other end of the workload spectrum, a single thread, an individual core should *scale up*, and pursue single-thread performance aggressively, at the cost of increased core power. Ultimately, to what extent cores should scale is a subjective decision: if performance is the *only* metric of value, cores should scale up fully. On the other hand, optimizing energy-efficiency may also be a motivating concern. Unfortunately, the most energy-efficient configuration is not immediately obvious, before runtime and energy profiling reveals which power/performance point is most suited to a particular workload. A hardware scaling policy can help to identify the most-efficient configuration (Section 6.4).

There are also workloads near the middle of the spectrum—e.g., those with significant parallelism but also occasional sequential bottlenecks. Scalable CMPs can optimize these executions in at least two ways. First, if sequential bottlenecks can be identified, *scaling up* the appropriate core can be used to partially ameliorate the bottleneck. Second, *scaling down* cores that spin or otherwise perform no useful work can improve efficiency by wasting less energy.

In pursuit of these optimizations, this chapter discusses several experiments concerning the design and use of scalable cores in chip multiprocessors. First, I describe how to leverage the Forwardflow core to implement dynamic scaling. Next, I consider *from where* cores should acquire their additional resources to implement scale up—e.g., from core-private resource pools (a *resource overprovisioning* philosophy) or from resources shared with other cores (a *resource borrowing* philosophy). I find that resource overprovisioning is preferable to borrowing, as cost of overprovisioning can be small (e.g., a 7% increase in chip area), and slowdown from borrowing resources from other cores can be significant (e.g., 9% on average). Overprovisioning also places fewer additional demands on system software, as cores can be scaled independently of one another without coordination.

Next, I consider the means by which hardware scaling policies, operating systems, and software can evaluate energy consumption (e.g., to make more informed efficiency-based scaling decisions). Section 6.3 presents a method by which to estimate the power consumed by a scalable core, with accuracy comparable to that of an architectural simulation [25]. This technique enables profiling *across* configurations, showing that *scalable core policies can measure the power consumption of each configuration* on a per-benchmark basis, with low overhead. Future software explora-

tion of policies for scalable CMPs can use simple models based on these findings to estimate per-configuration power consumption for a given workload.

After discussing *how* to scale cores, I next consider *when* to scale cores. Section 6.4 evaluates hardware scaling policies, which aim to optimize energy-efficiency of single-threaded applications. Section 6.5 proposes scaling policies in the context of multithreaded workloads. I find that programmer-guided scaling (e.g., in new applications) is highly effective at identifying and ameliorating sequential bottlenecks. When no programmer annotations are possible, spin-detection hardware often improves energy-efficiency when used to opportunistically scale-down spinning cores, thereby saving power at little or no performance cost. Lastly, I find that, though scale-up policies are often energy-efficient in Forwardflow, heuristics based on lock acquisition do not behave as intuition might suggest, for two reasons. First, critical sections do not necessarily constitute sequential bottlenecks. Second, hardware may be subject to false positives, leading to inaccurate identification of critical sections.

Lastly, this work explores methods by which future researchers can evaluate scalable CMPs without time-consuming simulation. In Section 6.6, I discuss the conditions under which *dynamic voltage and frequency scaling (DVFS)* can be used to approximate the general behavior of a Forwardflow scalable core. Together with the work on power estimation, this lays the groundwork for software-scale policies for future scalable CMPs.

Granularity. This work considers dynamic core scaling at fine time granularity, i.e., more frequent than timer interrupts implementing an operating system's time slice. Two underlying considerations motivate this choice. First, this work makes the initial contributions necessary to use non-simulation techniques to evaluate scalable cores at coarser granularity (e.g., by using DVFS

for performance, Section 6.6, and simple models for power, Section 6.3). Second, practical limitations of simulation-based evaluation limit the scope of this evaluation to what can be simulated in reasonable time. This constitutes approximately one scheduling slice for a modern OS, about 10ms of target machine time. I believe future work in this area should investigate policies for scalable work at coarser granularity, using other means.

Power Consumption. I have explicitly chosen *not* to set a fixed power budget for the simulated CMP used in this study. Instead, I evaluate power and performance in the context of efficiency metrics, $E \cdot D$ and $E \cdot D^2$. These metrics implicitly consider power: E is the energy cost of an execution, and D represents its runtime. The relative importance of E and D are represented as geometric weights in the measures—other weights are possible. Generally, I assume that a sufficient performance gain will motivate additional energy costs (i.e., power supply and heat dissipation).

6.1 Leveraging Forwardflow for Core Scaling

Chapter 5 discussed how to build a Forwardflow core with a statically-sized instruction window (in Forwardflow parlance, a *Dataflow Queue*, or *DQ*). This section addresses how to build a *dynamically scalable core*, using the Forwardflow static design as a starting point. Not every detail of Forwardflow core operation is pertinent to this discussion. Importantly, larger (or smaller) Forwardflow cores can be built simply by statically provisioning more (or less) DQ capacity, with no change to the remainder of the pipeline. In other words, enabling (or disabling) portions of the DQ effectively scales a Forwardflow core.

Three features of the Forwardflow design make it amenable to dynamic scaling. Firstly, the DQ is already disaggregated and organized into discrete subsections, *DQ bank groups* (*BGs*). The number of BGs in each design is the principal difference between the static design points presented in

Chapter 5, e.g., *F-128* is built from one group, *F-1024* uses 8[1]. Because window scaling affects core-wide (indeed, chip-wide) power consumption, scaling the DQ can be used to manage overall core power/performance. In other words, *scaling up means dynamically adding more bank groups* to a computation.

Secondly, the use of pointers throughout the design enables Forwardflow's control logic to handle window size reconfiguration gracefully—pointers are already oblivious of the physical characteristics of the structure to which they point. Since the DQ is managed as a FIFO, it is a simple matter to modify the head and tail pointer wraparound logic to accommodate variable DQ capacities that are powers of two, using modulo-2^N logic.

This leads to the third attractive feature, that Forwardflow-based scalable cores can be reconfigured relatively quickly—on the order of the time required to resolve a branch misprediction. Forwardflow offers a highly scalable window, and by altering head/tail wraparound conditions, the effective size of this window effectively scaled, affecting the behavior of the whole core. All that is needed to implement a change in window size is a pipeline flush.

6.1.1 Modifications to Forwardflow Design to Accommodate Scaling

When scaled up, a dynamically scalable Forwardflow core uses many bank groups to implement a larger instruction window. When scaled down, fewer BGs operate, thereby throttling overall core performance and power consumption. Because many possible operating ranges exist, the interconnect between dispatch/commit logic and the BGs, and between BGs themselves (i.e., the operand network) must be constructed with dynamic scaling in mind.

1. The number of checkpoints for the Register Consumer Table (RCT) varies per design as well, in support of larger windows.

To conserve leakage power, each bank group could be placed in a separate voltage plane, enabling an entire BG to be completely powered down. However, recall from Chapter 5 than individual bank groups are fairly small (less than $1mm^2$), and account for a relatively small amount of chip-wide leakage power (about 12% of chip-wide static power, or about 4% of power overall). Therefore, to enable greater flexibility in fine-grain scaling control, I assume unused BGs are quiesced (i.e., consume no dynamic power) but remain powered, so that they can resume operation quickly when scaling up.

Scaling Down. Scaling down is simple by comparison to scaling up. One could rely on simply bounding DQ allocation to a few entries, effectively constraining total window size. However, an opportunity exists to also scale down the effective pipeline *width*—offering a wider power/performance range (e.g., *F-32* in Chapter 5). To implement a power-performance point similar to that of the smallest static Forwardflow configurations (e.g., *F-32*), execution logic need only restrict the number of active banks within a single bank group.

Banks *within* a bank group are interleaved on low-order bits of the DQ identifier. I exploit this fact to dynamically scale down pipeline width. Instead of constraining DQ entry allocation when scaled down, dispatch logic instead advances the tail pointer by two, instead of one, after each allocation. Doing so "skips" individual DQ banks, effectively disabling one half of the DQ banks within a group (i.e., *F-64*), and forcing a dynamically narrower pipeline *and* a smaller window. The same approach can be applied to disable 75% of the DQ banks, by advancing the tail pointer by four entries on each allocation (i.e., *F-32*).

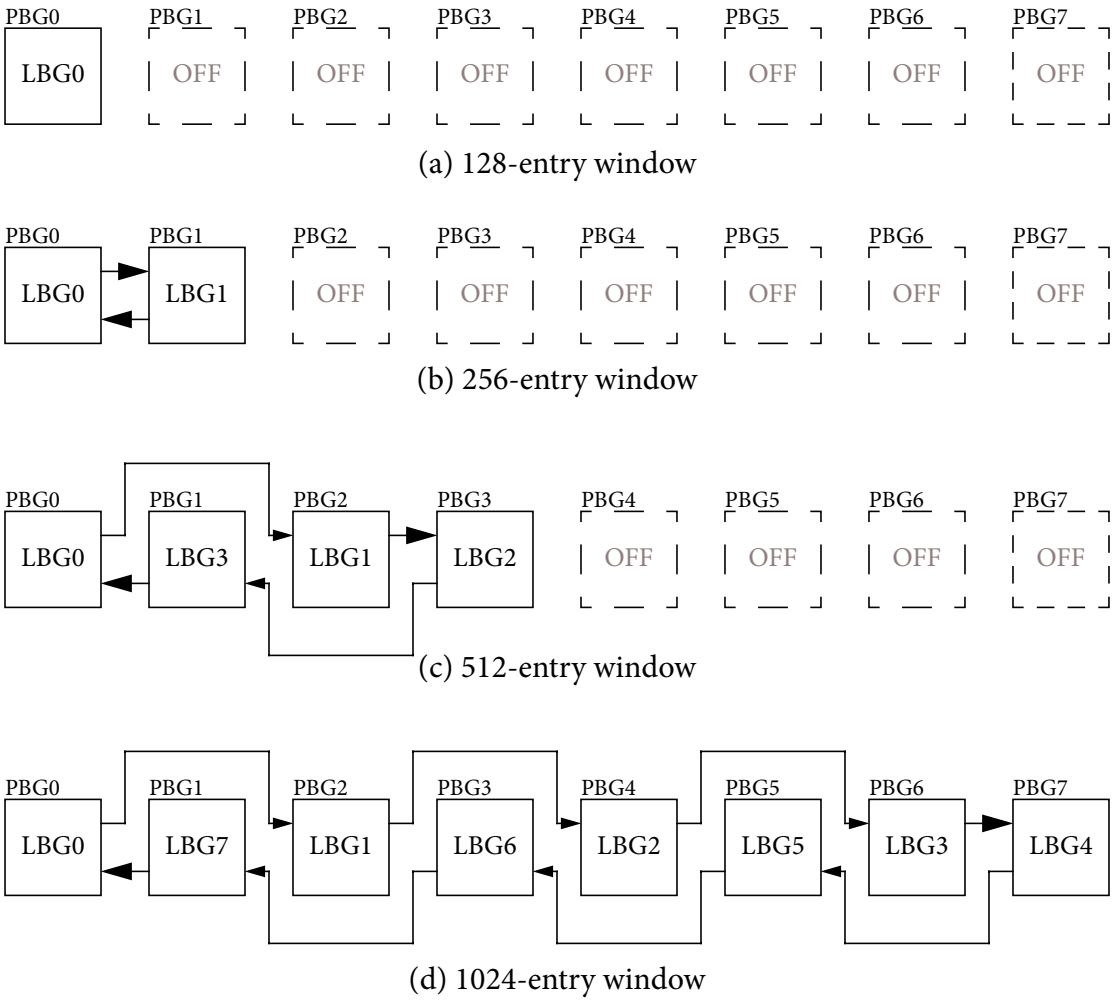


FIGURE 6-1. Logical connections between physical bank groups (PBG), 128-entry through 1024-entry Forwardflow windows, and associated logical bank group (LBG) mapping.

Scaling Up. Implementing dynamic scale-up requires some additional hardware, compared to a static Forwardflow core. This hardware involves the intra-core interconnects, which must be designed with the expectation that not all endpoints will be operating at a time. In particular:

- The dispatch bus must be routed to any BGs capable of operating with the dispatch logic (one at a time), though not all bank groups will actually operate when scaled down.

- Similarly, the commit bus must be routed from all BGs to the ARF control logic. As with the dispatch bus, only one group will commit at a time, and not all BGs will operate when scaled down (this bounds reconfiguration time to a few cycles).
- As was the case with large static cores, bank groups must operate collectively to resolve branch mispredictions.
- The operand network *between* BGs should bypass unused bank groups when not fully scaled up, to minimize inter-BG latency (on the critical path of instruction wakeup in Forwardflow).

Importantly, proper manipulation of DQ head and tail pointers' wraparound logic solves the first two considerations. In particular, if physical DQ entries backed by unused BGs are never actually allocated to instructions in the first place, window size is effectively constrained, requiring no change to the dispatch and commit logic used in Chapter 5. The remaining two interconnects are addressed with circuit-switched interconnects (a narrow hierarchical crossbar for mispredictions, not discussed in further detail, and a wider ring interconnect for the operand network, discussed below).

Implementing a Dynamically-Scalable Unidirectional Ring. When fully scaled up (i.e., to a 1024-entry DQ), a core uses eight BGs (each backing 128 contiguous DQ entries). Static Forwardflow uses a unidirectional ring between bank groups to implement inter-BG communication. Skipping every other BG in the ring homogenizes per-link latency[2] by avoiding a costly wraparound link. To implement this topology dynamically, the hardware implementing a particular bank group, i.e., the *Physical Bank Group* (PBG), is disassociated from the *Logical Bank Group* (LBG), the subset of the logical DQ space backed by a particular bank group. Figure 6-1 illustrates

2. Recall that each bank group is less than 1*mm* square. Each link incurs approximately 0.l9 *ns* signalling time.

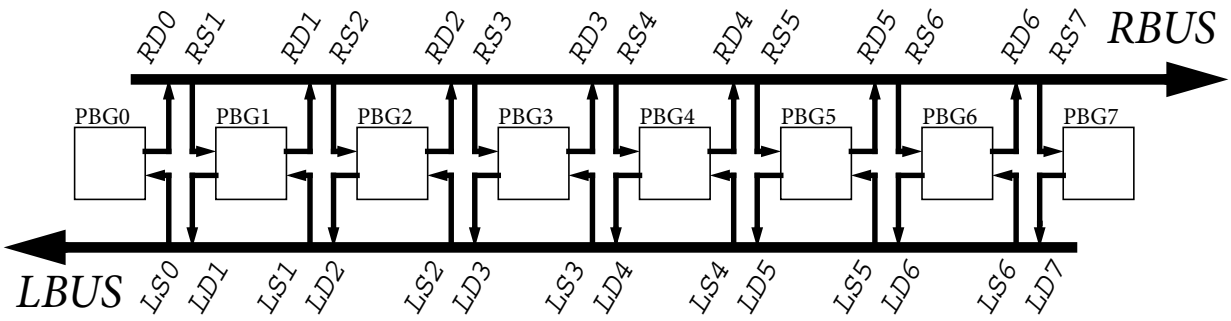


FIGURE 6-2. Dynamically circuit-switched unidirectional ring operand network.

the interconnect between bank groups, and corresponding PBG-to-LBG mappings for (a) 128-entry, (b) 256-entry, (c) 512-entry, (d) and 1024-entry windows.

I first considered simply statically implementing the 1024-entry case, and requiring unmapped LBGs to repeat all messages when scaled down. However, this practice effectively introduces a longer-latency wraparound link, consisting of unused PBGs, to complete the ring, thereby overly penalizing operand transfers from the last LBG the first. To address this problem, I propose a *dynamically circuit-switched operand network*, illustrated at a high level Figure 6-2, with per-BG connections in Figure 6-3. This interconnect consists of two unidirectional circuit-switched segmented busses: *RBUS*, propagating signals left-to-right, and *LBUS*, propagating signals right-to-left. Physical bank groups selectively connect to *RBUS* and *LBUS*, according to the contents of bus control words governing connections. In particular, each bus has a *drive control word* (i.e., `RD0`, "right drive zero", through `RD7` and `LD0`, "left drive zero", through `LD7`) and a *snoop control word* (i.e., `RS[7:0]` and `LS[7:0]`). Figure 6-3 depicts the logic to implement circuit switching for one bank group, omitting repeater logic for clarity[3].

3. The aforementioned 0.19*ns* delay used two repeaters per link.

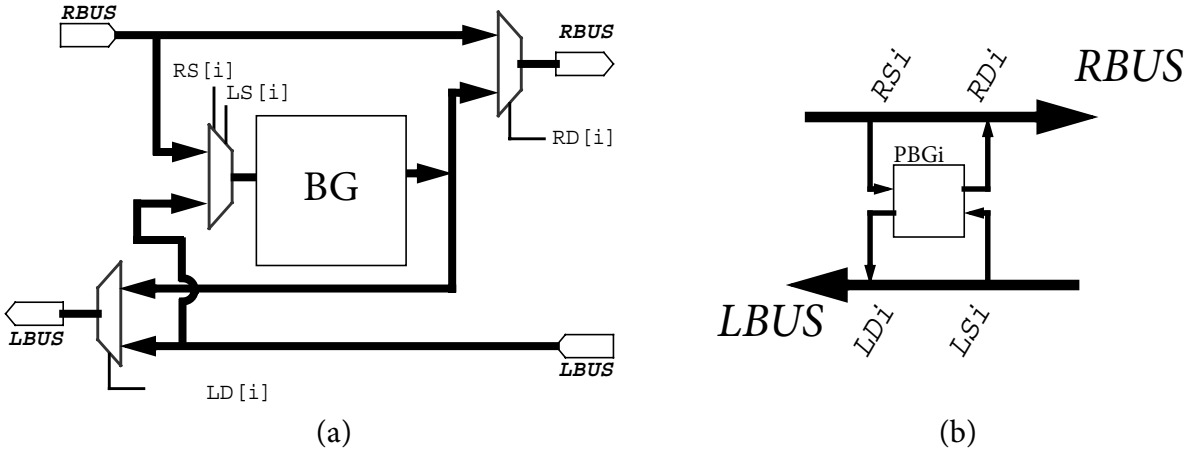


FIGURE 6-3. (a) Circuit-switched interconnect harness for a single bank group, and (b) circuit symbol for harness and BG, connected to *LBUS* and *RBUS*.

Under this organization, asserted bits in a bus's drive control word multiplex signals from the local bank group onto the bus. In other words, setting bit `xDi` connects segment `i` of bus `x` to the output port of bank group `i`, masking any signal behind position `i` such that it no longer propagates on bus `x`. Similarly, asserted snoop bits connect a bus to a local bank group's input port.

The LBUS/RBUS interconnect enables a variety of topologies, including those depicted logically in Figure 6-1. Figure 6-4 shows the interconnect scaled to the 256-, 512-, and 1024-entry design points, along with the associated control words for *RBUS* and *LBUS* connectivity. Single-PBG operation (for *F-128* and smaller configurations) is handled as a special case, in which the operand network does not operate at all. Greyed components are de-activated, dotted components are activated but unused. The most-significant bits of `RD` and `LS` are X, as *PGB7* has no right-hand connections to *RBUS* or *LBUS*, and similarly the least significant bits of `RS` and `LD` are X, as *PGB0* has no left-hand connections. Note that, because only a few unidirectional ring configurations are needed, the full generality of the LBUS/RBUS approach is not strictly necessary (e.g., several bits

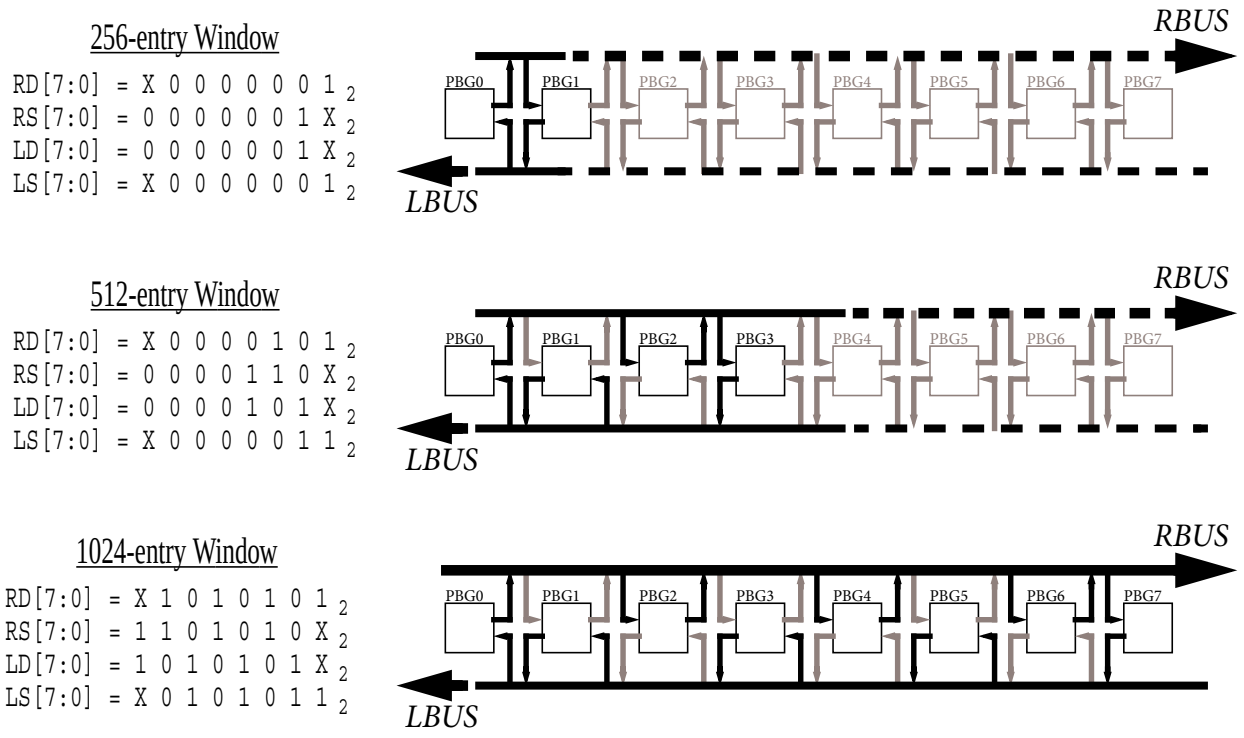


FIGURE 6-4. 256-, 512-, and 1024-entry window configurations of the dynamically-scalable interconnect, and associated values of *RBUS*/*LBUS* control words.

of the above matrixes are constant). However, implementing these optimizations would require custom logic at each PBG. It is simpler to implement all PBGs identically, and never activate unneeded paths during normal operation.

Lastly, note that the *RBUS/LBUS* implementation above is valid for either the resource borrowing or resource overprovisioning case (Section 6.2). In the former case, the *RBUS/LBUS* control words are shared between a collection of cores that could potentially share PBGs, potentially complicating the process by which control words are updated during reconfiguration.

The effective timing of the RBUS/LBUS implementation is no different than that of a static core, once circuit-switched paths are established. Unless otherwise noted, I assume wire delay (not logic delay) dominates the interconnect.

Software Considerations. The details of the underlying microarchitecture should not be exposed to software, as the next hardware generation's implementation may differ. Instead, there are several abstractions by which a scalable Forwardflow core could be presented to the system software. For instance, scalable core configurations can be abstracted as a power-control state [85] (e.g., in ACPI, a *negative* C-state). The precise mechanism used will depend on a variety of factors, e.g., whether cores scale dynamically in hardware (Sections 6.4 and 6.5), or strictly under software control. In the former, it seems reasonable for software policies to determine base and bound configurations, which can be easily abstracted as privileged registers. In the latter, a single register could be used to determine overall core configuration.

Reconfiguration. A reconfiguration operation should require only a few cycles to complete, given the above assumptions (circuit-switched operand network, DQ head and tail pointer manipulation to control allocation, and quiesced, but powered, bank groups). Reconfiguration latency should be only slightly longer than the recovery time from a branch or disambiguation misprediction. During reconfiguration operations, architectural state (i.e., the ARF, PC, and NPC) remains unchanged, all in-flight instructions are flushed, and instruction fetch is halted until reconfiguration is complete.

TABLE 6-1. Forwardflow Dynamic Scaling Configurations

Configuration	Description
F-32	**Fully scaled-down.** Single-issue Forwardflow core, 32-entry DQ bank.
F-64	Dual-issue Forwardflow core with a 64-entry DQ (two 32-entry DQ banks).
F-128	**Nominal operating point**. Quad-issue Forwardflow core with a 128-entry DQ.
F-256	Quad-issue Forwardflow core with a 256-entry DQ (two *F-128*-style DQs, organized into two bank group)
F-512	Quad-issue Forwardflow core with a 512-entry DQ (four bank groups).
F-1024	**Fully scaled-up.** Quad-issue Forwardflow core with a 1024-entry DQ.

Configuration Space. The remaining sections of this chapter assume six possible configurations in their Forwardflow scalable cores. These configurations are summarized in Table 6-1.

6.2 Overprovisioning versus Borrowing

The first area of evaluation in this chapter addresses the means by which individual cores acquire additional resources to implement scale-up. Broadly speaking, these additional resources can either be *overprovisioned*, on a per-core basis for exclusive use, or *borrowed* from other cores dynamically. The high-level tradeoffs of borrowing versus overprovisioning are more fully discussed in Chapter 2. In this section, I address the quantifiable costs of overprovisioning and borrowing as they pertain to a scalable Forwardflow-based CMP.

At the heart of this dichotomy is a simple tradeoff. Intuitively, per-core overprovisioning requires additional chip area, which is unused when individual cores are scaled down. On the other hand, resource borrowing may waste less area, but may also incur longer delays between scaled components, due to global wiring (e.g., when communicating with resources nominally

belonging to other cores). Borrowing may also complicate scale-up, as individual core scale-up must be coordinated among cores within the borrowing domain.

The Forwardflow design is amenable both to borrowing and overprovisioning. Since not all parts of a Forwardflow core scale dynamically (i.e., only the bank groups), overprovisioning cost is smaller than that of some prior work [86]. However, borrowing is also simplified, because only bank groups must reside near one another in the floorplans of neighboring cores (e.g., in Figure 6-5). No other components of the Forwardflow design scale up, so only bank groups must incur the complexities of borrowing.

Area Cost of Overprovisioning. Scaling *all* components of a core requires significant area [86, 94]. However, Forwardflow, and at least one other scalable core [180], implement scale-up using small, modular components. For these designs, the on-chip area occupied by scalable components is small in context of the entire die area.

In Forwardflow, the unit of scaling is a *DQ bank group* (BG). Each BG occupies about *0.9* mm^2 in *32nm* (Chapter 5). In context of other core components, the unscaled Forwardflow frontend and ARF occupies about *4* mm^2, and the L1-D cache occupies about *4.7* mm^2 [149]. The smallest Forwardflow cores used in this study occupy about *10.5* mm^2, the largest *16.7* mm^2. This suggests about *6* mm^2 of unused area when a fully (over-)provisioned core is scaled down, about 35% of the total per-core area.

When these overheads are considered at the scale of a complete CMP, the area cost of overprovisioning is less significant. Each bank of an 8MB shared L3 cache is *2.33x4.15 mm*. Per-core pri-

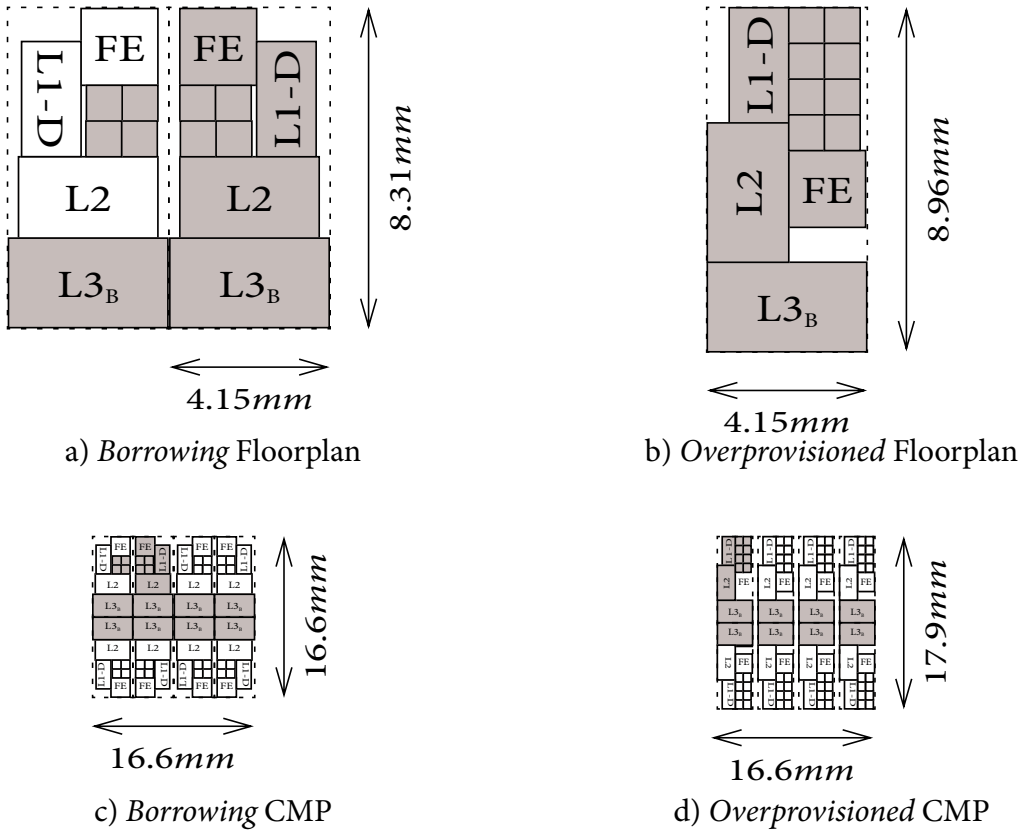


FIGURE 6-5. Forwardflow floorplans for core resource borrowing (a), per-core overprovisioning (c), a resource-borrowing CMP (c), and an overprovisioned CMP (d).

vate L1-D and L2 caches are *1.55x3.00 mm* (*4.7 mm*^{2})and *2.12x3.63 mm* (*7.7 mm*^{2}), respectively. Figure 6-5 plots these elements at scale ($1mm = 0.2in$ for core floorplans, $1mm = 0.05in$ for CMPs). "FE" indicates a core frontend. Floorplans a) and c) assume that each individual core includes enough bank groups to scale to *F-512*, and scaling to *F-1024* is implemented with borrowing. Floorplans b) and d) are fully overprovisioned—each core can scale to *F-1024* without borrowing resources. Greyed areas indicate powered components when one (logical) core is fully scaled up.

Per tile-area is about 8% larger in the overprovisioning case (*37.2 mm*^{2}, as opposed to *34.5 mm*^{2} in the borrowing case). This in turn leads to a slightly larger die size from an overprovisioned

design—the overprovisioned CMP is *298 mm^2*, whereas the borrowing-based CMP is *276 mm^2*, 7% smaller.

Though this analysis has assumed no area cost for borrowing, prior work has shown that implementing borrowing incurs its own area overheads. Ipek et al. estimate *the logic required to fuse cores occupies area comparable to that of an entire core* [86], about 12% more die area, most of which falls out from the complexity of wiring and logic needed to dynamically share components. These overheads would be substantially lower under an overprovisioning philosophy, as the resulting wiring would be core-local. This suggests that *it may be more area-efficient to overprovision*, as borrowing's routing complexity is much higher overall.

Delay Cost of Borrowing. The floorplans in Figure 6-5 show that the on-chip signalling distance between interconnected elements in the borrowing and overprovisioning scenario is comparable. However, this observation is specific to two-core sharing, and would not hold for larger borrowing pools. Moreover, addition of logic to enable dynamic borrowing may incur delay [86]. To evaluate the effect of this delay, I consider an experiment in which an additional two-cycle delay is added to signals that cross core boundaries. This delay affects the inter-BG interconnect, as well as the dispatch and commit logic. For simplicity, I do not model additional delays in branch resolution—these should be second-order effects.

Figure 6-6 plots the runtime of the fully scaled-up borrowing design (i.e., *F-1024*, in which the final scaling interval is implemented with borrowing) in grey, compared to the runtime of the next lower scaling point, *F-512* (dark grey), implemented with core-private resources. Both runtimes

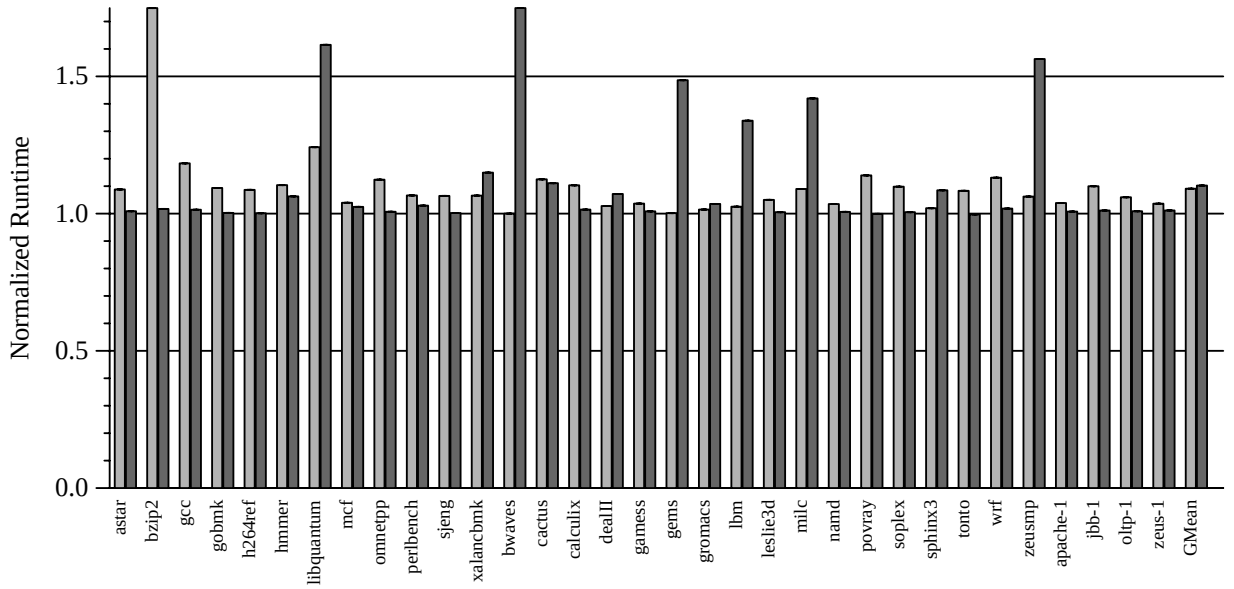


FIGURE 6-6. Runtime of *Borrowing F-1024* (grey) and *No-Borrowing F-512* (dark grey), normalized to that of the *Overprovisioned F-1024* design.

are normalized to that of a fully scaled-up *overprovisioned* design (i.e., an *F-1024* design like that of Figure 6-5 b). In *all* cases, the *added communication delay from borrowed resources reduces performance*, by an average of 9%. This means that the performance of the borrowed design, fully scaled, is not much faster on average than one core operating alone, *F-512*. With only small delays between core domains, there is little point to ever scaling to *F-1024/Borrowed*.

Two root causes underlie the performance gap between borrowing and overprovisioning. Firstly, dispatch to borrowed resources is slower, incurring a longer effective fetch-to-dispatch latency. Second, pointer/value pairs traversing the inter-bank-group interconnect experience delays when crossing core boundaries. To address the effect of the latter, Figure 6-7 plots the executing occupancy of the same designs. Recall that the executing occupancy is the number of instructions in the window that have been scheduled but not yet completed. In other words, it measures the mean number of instructions simultaneously “executing” in a design, rather than waiting to execute, or waiting to commit. This measure is again normalized to that of the overpro-

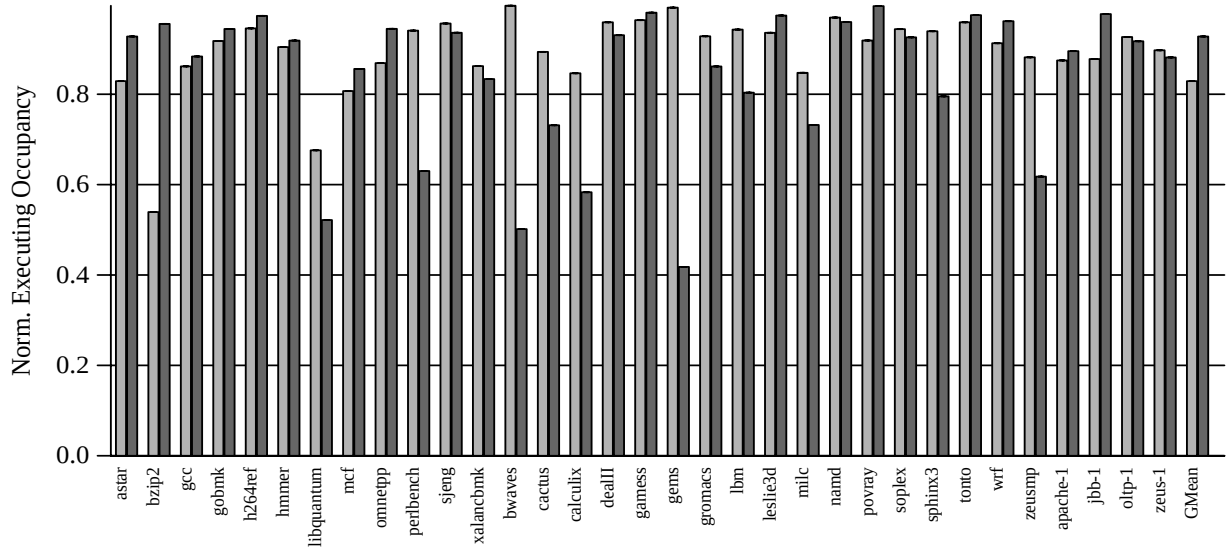


FIGURE 6-7. Executing occupancy of *Borrowing F-1024* (grey) and *No-Borrowing F-512* (dark grey), normalized to that of the *Overprovisioned F-1024* design.

visioned *F-1024* design. Overall, added delays in the inter-BG network tend to reduce executing occupancy, as wakeups tend to be delayed when they (unluckily) cross a core boundary. These extra delays never occur in the core-private *F-512* (dark grey), or the fully-overprovisioned baseline.

Despite operating with slightly lower performance, the borrowing case exercises the unscaled hardware similarly to the overprovisioned case, resulting in similar overall power consumption. *Comparable* power and *worse* performance implies *lower* energy-efficiency from overprovisioned designs. Figure 6-8 plots efficiency ($E \cdot D$, top, and $E \cdot D^2$, bottom) of *Borrowing F-1024* and *No-Borrowing F-512* with respect to the overprovisioned design. Overprovisioning is never more efficient than borrowing, and is usually substantially less efficient. On average, it is more efficient to simply not scale beyond *F-512*, if borrowing is needed to do so.

While this experiment considers only the effect of borrowing to scale to the largest configuration, i.e., from *F-512* to *F-1024*, a larger lesson remains: borrowing will hurt performance and effi-

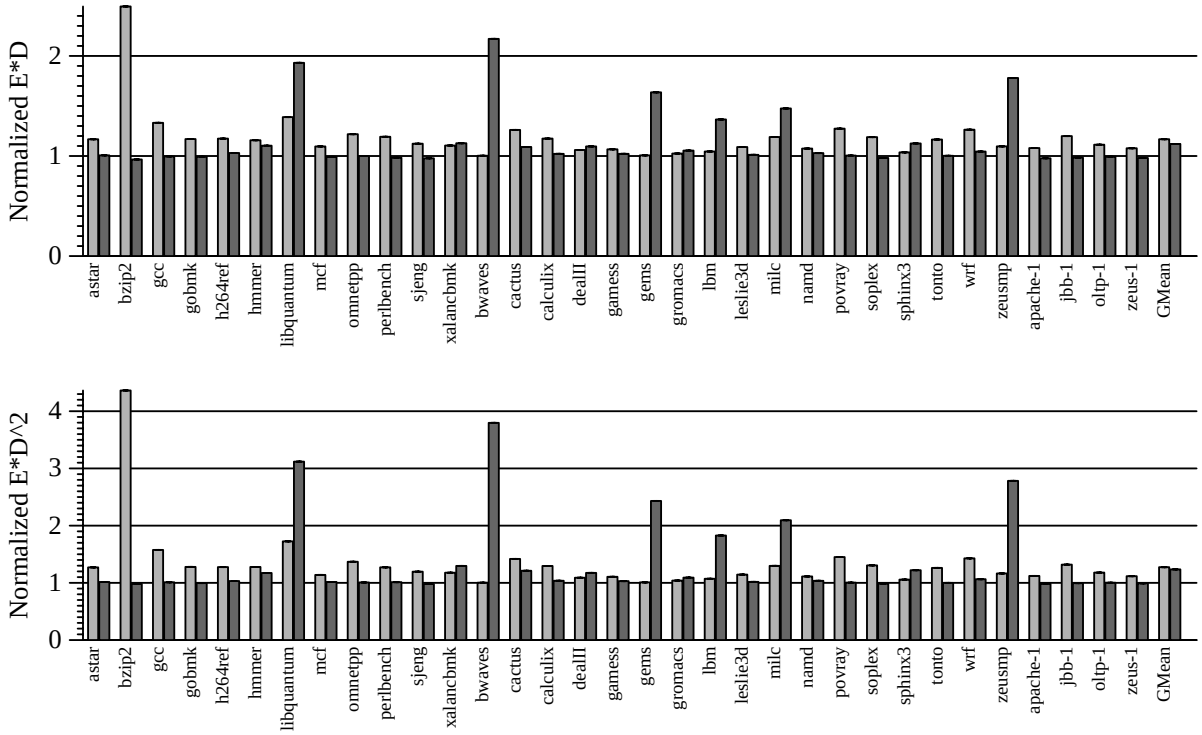


FIGURE 6-8. Energy efficiency of *Borrowing F-1024* (grey) and *No-Borrowing F-512* (dark grey), normalized to that of the *Overprovisioned F-1024* design.

ciency if it introduces more delay than an overprovisioned design. Though this experiment concentrates on the last stage of scale-up, if borrowing is used at lower levels of the configuration hierarchy (e.g., from *F-256* to *F-512*), one can expect worse results. Any borrowing-induced delays will occur more frequently in these smaller core designs, and will even more severely affect performance. On the other hand, the area benefit of borrowing (if one exists) will see diminishing returns as scalable components represent progressively less per-tile area.

Summary. While prior work has implicitly advocated resource borrowing, I show that if borrowing introduces delays, *overprovisioned* Forwardflow cores are likely to perform better. The cost of implementing fully-overprovisioned cores is fairly small: die area is increased by only 7% in the hypothetical CMP used in this study. I conclude that, so long as per-scaled component area is small, future designs should simply overprovision cores with sufficient resources to scale up,

rather than complicate hardware and software design with coordinated inter-core resource borrowing.

6.3 Measuring Per-Core Power Consumption

Future hardware and software policies for scalable CMPs must seek a balance between power consumption and performance demands. Performance is usually a straightforward metric: wall clock time or throughput are both easily measured. But measuring power consumption is more difficult, especially at fine granularity. To enable on-line estimation of energy-efficiency, scaling policies and heuristics for future scalable CMPs need a means by which to reliably estimate per-configuration and per-benchmark power consumption.

The power consumption of a circuit can be measured externally, e.g., by measuring a voltage differential across a shunt resistor [89]. However, circuit-level techniques to estimate power require significant additional hardware, are generally coarse-grained (e.g., measure the consumption of an entire voltage plane, or an entire chip), and measure energy consumption over long time intervals (i.e., on the order of milliseconds). These characteristics make circuit-level measurement undesirable for use in scaling heuristics, for three reasons. First, hardware dedicated to scaling heuristics should be small, non-invasive, and have low cost. Second, fine-grained scaling heuristics require fine-grained estimates of energy consumption, preferably estimates that can distinguish between consumption affected by scaling (e.g., the power consumed in the instruction window and datapaths) and consumption independent of scaling (e.g., leakage current and clock power). Third, and perhaps most important, scaling heuristics will use estimates of energy consumption to evaluate which scaling points are currently preferable, given the current behavior of a workload. Therefore, estimates should be made over relatively short periods of time—it may be unacceptable

to evaluate a particular scaling point for a few tens of milliseconds, as an entire phase of the computation can elapse in that time [147, 148, 172].

Precise estimation of power consumption allows a scaling heuristic to operate with knowledge, rather than assumption, of actual energy consumed by a calculation, leading to clearer power/performance tradeoffs. This subsection develops a method by which scaling heuristics or system software can estimate per-configuration power consumption, based on easily-observable microarchitectural event counts. This feature enables a hardware scaling policy to directly measure energy consumption, with minimal core hardware modification, or for software to estimate energy directly by executing a small subroutine. Importantly, this approach requires almost no new hardware, as it uses the core's own execution resources to compute energy estimates.

Some prior work in this area has proposed or assumed specialized hardware to provide runtime power estimation [87, 89], rather than direct physical measurement. The premise behind these proposals is the same used by Wattch [25], a simulation tool for architectural-level approximations of power consumption in CMOS. Wattch operates on the simple assumption that all microarchitectural events have a fixed energy cost—by construction the entire energy cost of an execution is the sum of the products of counts of microarchitectural events and their associated energy cost. In other words:

$$E_{execution} = \sum_{i \in A} Activations_i \cdot E_i \tag{6.1}$$

where A is the set of all microarchitectural components, $Activations_i$ represents the number of activations of each component i in the execution, E_i is the energy consumed by an activation of component i. $E_{execution}$ is the total energy consumed during the execution.

However, though Wattch is a simulation tool, the same insight applies to online measurement of actual systems. Though not all $Activations_i$ can be easily determined at runtime, Joseph et al. [89] show that $Activations_i$ can be predicted with high accuracy from software-observable performance counters in the Alpha 21264 and the Pentium Pro, based on a linear cost model. In other words:

$$E_{execution} \approx E_{est} = \sum_{j \in C} N_j \cdot E_j^{est} \tag{6.2}$$

where C is the set of *observable* events, N_j is the count for each event, and E_j^{est} is a linear weight (nominally, an energy cost) assigned to each event type.

Equations 6.1 and 6.2 differ in several key areas. First, Eqn. 6.2 is approximate—it relies on precise measurement of event counts N_j and accurate per-event linear weight E_j^{est}. Second, the set C consists of microarchitectural events that can be easily exposed as a performance counter (e.g., a count of decoded instructions, or the number of local cache accesses), whereas the set A (in Eqn. 6.1) consists of *all* possible microarchitectural events, including those not easily made available to performance counters (e.g., the number of remote cache invalidations). A tradeoff arises because the accuracy of the latter method improves as C becomes more inclusive of A, but the complexity of implementing the sum in Eqn. 6.2 grows with the cardinality of C. In other words, for accuracy,

it would be preferable to measure *all* microarchitectural events, but for practicality, measurement must be restricted to those events that are easy to count at a single processor core.

Fortunately, various N_j are predictably correlated to various $Activations_i$. For example, for each instruction committed (a measure trivial to include in set C), one can infer that the instruction was fetched (i.e., the fetch logic was activated), the instruction was decoded (i.e., microcode tables were consulted), source operands were read (i.e., the register file was read), etc. These correlations can be measured, to assign E_j^{est} in such a way that the error of Eqn. 6.2 is minimized.

The problem of finding an acceptable set of countable events C is that of a minimization problem: *pick the smallest set of counters possible to minimize estimation error.* Table 6-2 enumerates 34 possible candidate event counts for set C, selected from events already present in the microarchitecture of a Forwardflow core.

TABLE 6-2. Microarchitectural events in a Forwardflow core (candidates for set C)

Event Name	Description
`l1d_misses`	Count of misses in the core's local L1-D cache.
`l2_misses`	Count of misses in the core's local L2 cache.
`l3_misses`	Count of misses in the L3 cache (from local core).
`dram_access`	Count of total DRAM accesses (from local core).
`mshr_recycle`	Number of MSHRs allocated.
`l1d_access`	Count of all L1-D accesses and D-TLB translations.
`l1i_access`	Count of all L1-I accesses and I-TLB translations.
`l1i_misses`	Count of instruction misses resulting in a stall condition.
`fetch_buffer_hits`	Count of instruction fetches residing on the same cache line as the previously-fetched instruction group.
`fetch_puts`	Count of cycles in which the fetch pipeline flows instructions to the instruction decoders.
`decode_puts`	Count of cycles in which the decode pipeline flows instructions to the dispatch logic.[a]
`bpred_cnt`	Count of predicted branches.
`total_instrs`	Count of total retired instructions.

TABLE 6-2. Microarchitectural events in a Forwardflow core (candidates for set C)

Event Name	Description
`cycle_count`	Count of elapsed processor cycles since boot.
`nosq_predictions`	Count of predictions made by memory dependence predictor.
`nosq_faults`	Count of incorrect predictions made by memory dependence predictor.
`load_instrs`	Count of committed load instructions.
`store_instrs`	Count of committed store instructions.
`atomic_instrs`	Count of committed hardware atomic instructions.
`load_miss`	Count of load instructions that miss in the L1-D cache.
`store_miss`	Count of store instructions that miss in the local L2 cache.
`ialu_ops`	Count of committed instructions requiring an integer ALU.
`falu_ops`	Count of committed instructions requiring a floating point ALU.
`dq_ht_count`	Count of activations of the DQ's head and tail pointer logic.
`uu_b0_count`	Count of activations of update hardware on bank 0 of the DQ.
`bg_trans_cnt`	Count of operand transfers between bank groups.
`arf_writes`	Count of writes to the ARF.
`dflow_limit_cycles`	Count of cycles commit stalls with a non-empty DQ.
`dq_full_cycles`	Count of cycles dispatch stalls due to a full DQ.
`decoded_isntrs`	Total count of decoded instructions.
`rct_count`	Count of writes to the top-level RCT.
`rct_chckpt_count`	Count of RCT checkpoints created.
`rct_stall_cycles`	Count of cycles in which decode stalls due to a hazard in the RCT.
`contig_dispatch_cycles`	Count of cycles in which four instructions dispatch concurrently.

a. In practice, `fetch_puts` and `decode_puts` measure very different throughputs, as `decode_puts` encompasses throughput from micro-operations as well as architectural instructions.

Given the 34 above counters, the number of all possible sets C is:

$$Sets_C = \binom{34}{|C|} \tag{6.3}$$

Obviously, a complete exploration of $Sets_C$ is infeasible for $|C| > 3$. To quickly explore the space of possible counters, I used a genetic algorithm to quickly search the space of possible combinations of counters, using heuristic-based hill-climbing to find a set of coefficients E_j^{est} such that error (i.e., $\sum_{k \in K} |E_{est-k} - E_{execution-k}|$) is minimized over several executions K.

TABLE 6-3. Goodness of fit to actual power consumption of SPEC CPU 2006 and Wisconsin Commercial Workloads for six Forwardflow configurations.

Forwardflow Configuration	R^2
F-32	0.9992
F-64	0.9989
F-128	0.9977
F-256	0.9954
F-512	0.9917
F-1024	0.9834

Executions K consisted of SPEC CPU 2006 and Wisconsin Commercial Workload benchmark suites, and Forwardflow configurations ranging in size from 32-entry DQs to 1024-entry DQs. For each configuration. I constrained the search criteria to consider only $|C| \leq 8$ to keep the number of event counters manageable. One degree of freedom was effectively removed, however, as all solutions were required to consider the `cycle_count` register as part of set C, as early experimentation revealed that no other count adequately accounts for static power consumption. Between configurations of a Forwardflow core, I allowed the minimization algorithm to select different coefficients E_j^{est}, but the set C was fixed regardless of configuration. This constrains the problem space to discover counters that must be exposed at design-time to the power estimation logic, but assumes that linear coefficients can be loaded selectively, based on configuration.

The genetic process identified `store_instrs`, `fetch_puts`, `mshr_recycle`, `l1i_access`, `l1d_access`, `ialu_ops`, `falu_ops`, and `cycle_count` (forced) as optimal C. For all configurations, coefficients E_j^{est} exist that yield an estimate of explained variance (goodness of fit) $R^2 > 0.98$. Results are detailed summarized in Table 6-3. These results indicate that the same event counts, with different linear weights, can be used to measure the energy consumed

by Forwardflow-style scalable cores on per-configuration and per-benchmark basis. Figure 6-9 exemplifies this finding, with a plot of estimated power *residuals* for one Forwardflow configuration (*F-128*), over the SPEC CPU 2006 benchmarks. The figure demonstrates very accurate overall power estimation derived from the eight selected event counts. The greatest relative error occurs in the power estimate of the `calculix` benchmark, an error of approximately -6%. The greatest overestimate is around 4%, and occurs for the `xalancbmk`. Root-mean-square error is 1.4%.

Overall, these results are encouraging. The tight R^2 and root-mean-square error measures indicate that the "true" power can be very closely approximated by a weighted sum of correlated events. To actually realize this functionality, Appendix B presents a SPARC-like instruction sequence that implements an algorithm to estimate energy consumption, using the eight counters identified by the genetic process.

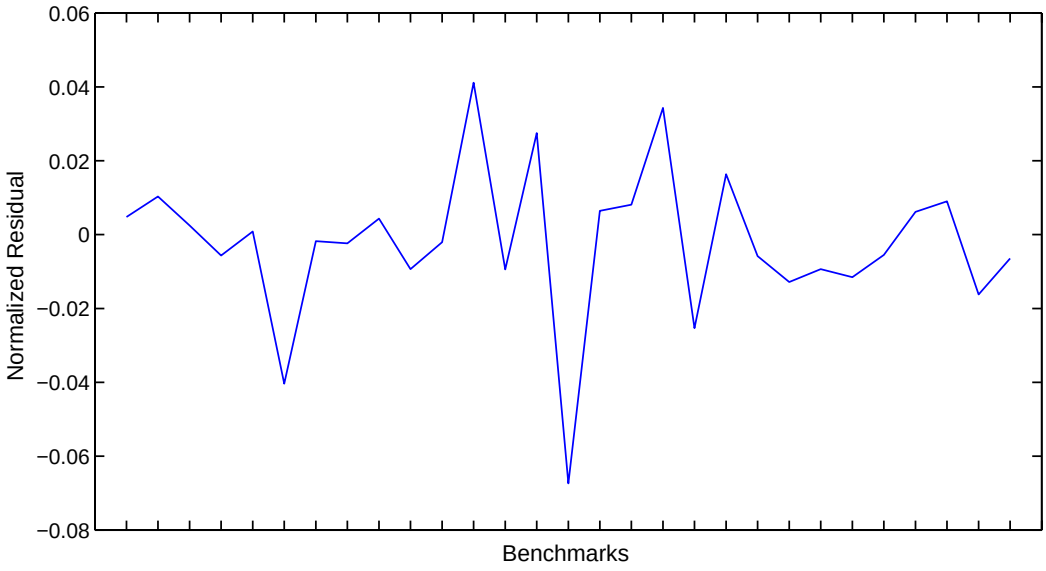


FIGURE 6-9. Normalized Residuals, power estimation versus observed power for configuration *F-128* over SPEC CPU 2006 benchmarks. Benchmarks are ordered as follows: SPEC INT (alphabetical), SPEC FP (alphabetical).

TABLE 6-4. Runtimes of power estimation algorithm on six Forwardflow configurations.

Forwardflow Configuration	Warm Cycle Count	Cold Cycle Count
F-32	528 ±1	1440 ±10
F-64	496 ±1	1407 ±10
F-128	484 ±1	1395 ±10
F-256	484 ±1	1395 ±10
F-512	484 ±1	1395 ±10
F-1024	484 ±1	1395 ±10

I evaluated the execution time required to produce an energy estimate by running the power estimation algorithm on each Forwardflow scaling point. Table 6-4 reports runtimes on each Forwardflow configuration. The algorithm uses a small, configuration-specific in-memory working set, which contains per-configuration coefficients E_j^{est}. Runtimes for in-cache (warm) runs and out-of-cache (cold) runs are shown in the table. Configurations *F-128*, *F-256*, *F-512*, and *F-1024* perform indistinguishably because of the extremely short number of executed instructions (the entire calculation fits within one *F-128* window, therefore the extra resources of larger configurations are never used).

Summary. This subsection shows that it is feasible for scaling heuristics to measure energy consumed by a each individual core. The estimation algorithm presented in Appendix B is feasible to implement, highly accurate (e.g., $R^2 > 0.98$) and has low overhead (e.g., approximately 500-1500 cycles, depending on cache behavior). This feature is usable by either hardware or software scaling policies, and does not require prior information about a workload—only prior information about dynamic configurations is required.

6.4 Fine-Grained Scaling for Single-Threaded Workloads

In this section, I consider the operation of a scalable CMP when running a single thread. System software may not know *a priori* which configuration of a scalable core is most efficient for a particular workload. I develop a fine-grained hardware scaling policy that can tune single cores' efficiency to match their workload, either as a proxy for the best static configuration, or to aid system software in discovering that configuration. Overall, the characteristics of the Forwardflow cores used in this study support the conclusion that fine-grained scaling policies can usually approximate the efficiency of the best static configuration, but only exceed the efficiency of the best static configuration when extensive profiling is available.

In general, a *hardware scaling heuristic* determines when a scalable core will change configuration, and to which configuration it will scale. I propose a scaling policy, based on observed memory-level parallelism (MLP), for determining optimal window size, an approach which works well with the dynamically-scalable Forwardflow cores used in this study. I compare this approach against two prior approaches, based on direct measures of efficiency. I find that MLP-based scaling usually achieves efficiency comparable to that of the best static configuration (which is not known *a priori*). If possible, pre-profiling and static annotation can sometimes further improve efficiency, for some workloads.

Scaling for Energy-Efficiency. If performance is the *only* concern, scalable cores should simply scale up, to the extent possible with available power. No sophisticated hardware policy is needed to support this decision: system software is fully aware of the number of threads running on a CPU, and can act accordingly.

However, when considering the possibility of scaling to optimize other metrics, e.g., energy-efficiency ($E \cdot D$ and $E \cdot D^2$ in this study), software-level scaling choices are less obviously clear. Section 6.3 suggests that software can profile workload power/performance tradeoffs, to selectively optimize whatever metric is desired, but software cannot easily adapt to workload conditions at extremely fine granularity (e.g., below the level of the smallest scheduling slice), and profiling runs are not always possible. Instead, a hardware scaling policy could be used to adaptively and opportunistically scale a core to optimize an efficiency metric, at almost arbitrarily small granularity.

For instance, it is intuitive that a large window may be efficient for one phase of an application but not another. Scaling affects efficiency in two ways. First, large windows do not show performance improvements if the underlying execution has little or no ILP and/or MLP. In such a circumstance, *scaling down* improves efficiency by conserving power without affecting performance. On the other hand, when a workload exhibits ILP, *scaling up* improves efficiency completing work faster, despite higher power consumption.

6.4.1 Single-Thread Scaling Heuristics

A scaling policy performs two basic functions: *decision*, the act of determining an appropriate configuration, and *indication*, the act of determining when to re-evaluate decisions. A variety of heuristics have been suggested in the context of dynamically reconfigurable hardware [9, 17, 19, 44, 45, 50, 65, 80, 81, 87, 129], covering a variety of scaled components (e.g., caches, pipelines, power-saving techniques), and a variety of optimized metrics (e.g., performance, efficiency metrics, power). This section considers representatives of these proposals, as well as a new scaling

heuristic based on MLP, to identify which general techniques are energy-efficient in the context of scalable cores running single threads.

Limitations of Local Decisions. Implementing *decision* is a formidable problem. In general, configuration choices that optimize energy-efficiency over one execution interval do not necessarily optimize larger encompassing intervals. Consider optimization of $E \cdot D$ (the same argument below applies to $E \cdot D^2$, or any $E^x \cdot D^y$). To find the optimal configurations over time, a heuristic must find a series of configurations S over indication intervals I such that:

$$E \cdot D = \left(\sum_{i \in I} E_{S_i} \right) \cdot \left(\sum_{i \in I} D_{S_i} \right) \tag{6.4}$$

is minimized. Among C configurations, there are $C^{|S|}$ possible configurations over time to consider which might minimize the sum in Eqn 6.4. Importantly, *selections S that individually minimize $E_{S_i} \cdot D_{S_i}$ are not guaranteed to minimize the overall sum.* However, scalable cores that usually exhibit power/performance monotonicity should not often expose this fallacy. That is, the most efficient configuration c for an interval i *usually* remains the overall optimal choice within a larger interval that encompasses i. Exceptions to this rule of thumb arise only when configurations expose non-monotonic extrema (e.g., a very slow but low-power configuration), suggesting a local pathology.

Positional Adaptation (*POS*). Huang et al. [81] suggest a "positional" approach to microarchitectural scaling, by associating a particular configuration with a static code segment in an application. In other words, the acts of *indication* and *decision* are based on which static program segment is executing. The authors explore a cross-product of static (i.e., compile-time) and dynamic (i.e., run-time) approaches to identify positional decision points, and to implement the decision algo-

rithm. The authors show that static profiling is desirable when possible; I evaluate positional adaptation in the context of static analysis only in this work, as other baselines adequately represent dynamic alternatives.

POS leverages pre-execution profiling and static analysis to implement both indication and decision. Profiling inputs are used for these runs, one run per configuration. After profiling, a post-processing step encodes configuration choices into the target binary. During profiling, energy and delay per subroutine is recorded, for use in post-processing.

In light of the difficulty of determining an optimal set of configurations s to minimize Eqn. 6.4, I consider two different algorithms for computing s from the profiling data. The first algorithm, *POS$*, greedily makes locally-optimal configuration selections over each interval. Its execution demands (on top of a profiling run for each configuration) are insignificant. I also consider *POSg*, which attempts to choose s using a genetic algorithm, using Eqn. 6.4 as its fitness criterion. I constrain the effort of *POSg* such that its runtime does not exceed the runtime of the benchmark itself—i.e., total profiling overhead is roughly bounded to a factor of the number of configurations plus one.

Figures 6-10 and 6-11 plot efficiency of *POSg*, with respect to that of *POS$*, for the integer benchmarks. The results of floating-point and commercial workloads are similar. In most cases, greedy and genetic optimizations yield similar overall efficiencies, with occasional exceptions due to local minima (e.g., `libquantum`). However, the additional computational demands of *POSg*

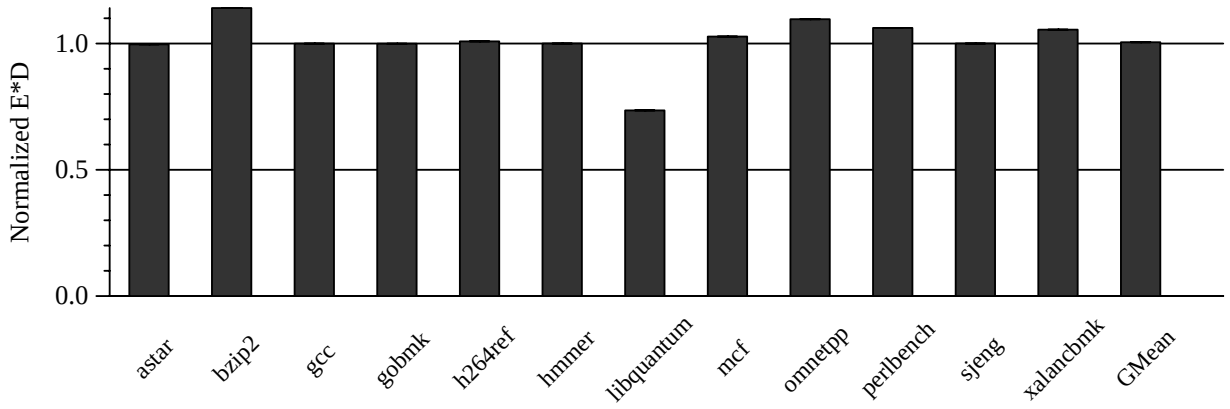


FIGURE 6-10. Efficiency ($E \cdot D$) of *POSg*, normalized to *POS$*, SPEC INT 2006.

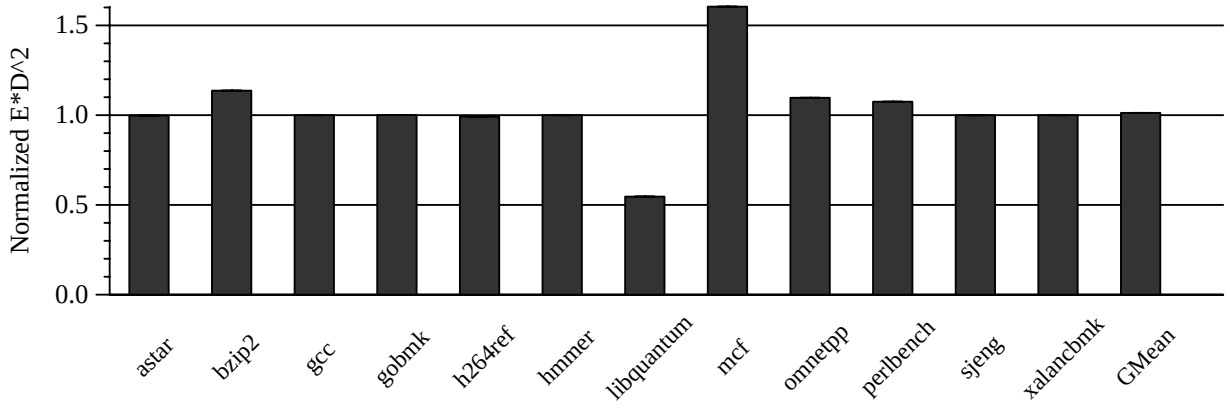


FIGURE 6-11. Efficiency ($E \cdot D^2$) of *POSg*, normalized to *POS$*, SPEC INT 2006.

are seldom worthwhile, as it seldom significantly improves efficiency with respect to *POS$* (which requires less pre-processing effort, including all profiling runs). This agrees with intuition suggesting that local energy-efficiency decisions are usually acceptable. In light of this result, I reduce the complexity of the subsequent evaluations by considering only *POS$*.

Power-Aware Microarchitectural Resource Scaling (*PAMRS*). Contrasting the static approach of *POS*, Iyer and Marculescu [87] suggest a dynamic approach to both indication and decision (others have also suggested a dynamic approach [19, 50, 81, 129]). Specialized hardware is used to implement *indication*, which retains a table of frequently-executed branches. This table

retains execution counts of control transfers, and is used to determine when the execution enters a new "hot spot", or more generally, superblock.

To implement *decision*, *PAMRS* simply walks the space of possible configurations, directly sampling energy and delay for each [87]. After sampling, the locally most-efficient configuration is selected until a new "hot spot" is observed. I adopt the ratio of sampling interval size to window size used by Iyer and Marculescu. This constitutes a sampling interval of 16x more instructions than can be held in the largest evaluated window (i.e., 16,384 instructions). I performed a brief sensitivity analysis of longer intervals, with little observed difference among six subset workloads.

MLP-Aware Window Scaling (*MLP*). *PAMRS* and *POS* take an efficiency-measurement approach to configuration management, via direct sampling and estimation, and are generally unaware of the specific effects leading to efficiencies (or in-efficiencies) resulting in its configuration decisions. Instead of direct sampling, I propose an approach in which the underlying characteristics of the scalable core's affinity (or lack thereof) to a particular workload directly influence *indication* and *decision*. For Forwardflow cores, this characteristic is memory-level parallelism (MLP).

I describe a method by which to determine the *smallest dynamic window size needed to expose all possible MLP*. Intuitively, the smallest window size that achieves all possible MLP should perform close to that of the largest window size. However, by using fewer resources, it should also save power, thereby optimizing (local) energy efficiency.

The hardware cost of determining the locally-optimal window size is fairly modest:

- Per-architectural-register *poison bits* identify independent operations,
- A per-instruction bit to indicate whether or not the instruction is a load miss,

- Ten bits per instruction in the DQ's metadata array, to retain architectural register names,
- A linear-feedback shift register (LFSR),
- A *commit counter* W with range [0, W_{max}), where W_{max} is the window size of the largest configuration, and,
- An *exposed load miss register* (*ELMR*), consisting of one bit per configuration. The highest configuration with a set bit at the end of profiling is the optimal window size.

Figure 6-12 depicts the MLP profiling process as a flow chart. To begin, the committed instruction stream[4] is decomposed into profiling intervals, approximately W_{max} in size. Profiling always begins on a load miss, randomly sampled using the LFSR[5]. When profiling, hardware counts the number of committed instructions in the interval (W) which is used to identify appropriate window sizes (described below). The profiling interval ends after W_{max} instructions commit.

In order to identify load misses during commit-time profiling, a single bit is retained (e.g., in the DQ metadata array) to indicate whether or not a load missed in the cache. To further identify *independent* load misses, profiling hardware maintains an array of poison bits, one for each architectural register (e.g., similar to those used in *Runahead Execution* [122])[6]. When a profiling interval begins, poison bits and the ELMR are cleared. The first committed load miss—i.e., the operation that triggered the profiling interval—sets the poison bit associated with its output regis-

4. Using committed instructions keeps the MLP profiling hardware off of the critical execution path.
5. Sampling must be randomized to prevent aliasing, in which the same load is profiled within a loop because of a coincidental relationship between W_{max} and the length of the loop.
6. There is no need to physically locate poison bits within the ARF.

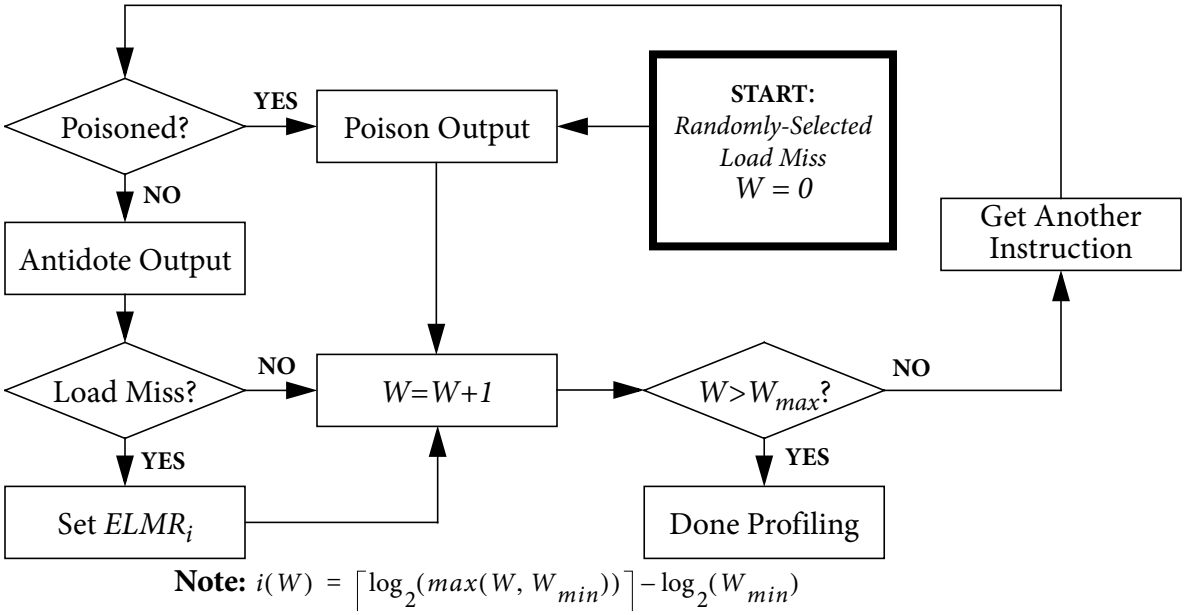


FIGURE 6-12. Profiling flow for MLP estimation.

ter. While profiling, committed instructions that name a poisoned register as an operand set the poison bit on their output register. Conversely, when an instruction commits with *no* poisoned operands, profiling logic clears the output register's poison bit. In this manner, poison bits propagate from the first load miss to all *dependent* committed instructions (and only dependent instructions) in the profiling window. When a load commits with no poisoned input operands, the load is data-independent of the load that initiated the profiling interval. Importantly, an *exposed miss* is detected if the committing load is also a load miss.

Observation of an exposed miss may cause a bit to set in the ELMR, depending on the current value of W (the count of the number of committed instructions since profiling began). Each bit $ELMR_i$ is associated with window size W_i: if $W \geq W_i$, $ELMR_i$ is set upon observation of an exposed miss. This hardware is trivially implemented with a *set-enable* register on each bit of the ELMR, tied to the appropriate carry signal in the commit counter W.

Hysteresis. At the conclusion of the profiling period, the most significant set bit in the ELMR indicates the configuration that would have exposed the greatest number of misses, had that configuration been in use when the profiled instructions executed. Because the ELMR-indicated size is a function of only a single load miss, the size reflects only what might have been achieved with respect to that load's forward slice, and not the best overall configuration. To provide an element of memory, by which many loads can be considered over time, only *upward* changes in configuration are accepted from the ELMR. In other words, profiling never directly causes a core to scale down. Over time, core configuration approaches the largest configuration that was *ever* useful, rather than the largest configuration that was useful for the most recent profiled load miss. To force workloads to re-assert value of larger configurations, cores scale down (fully) whenever a full-squash condition is encountered (e.g., an external interrupt or exception), so that an application must re-exhibit affinity for larger windows.

The need to limit downward scaling from MLP estimation was not immediately apparent. Figure 6-13 plots the configurations selected by the MLP heuristic, as a function of position in the benchmark (measured in instructions committed), for two selected benchmarks. Specifically, the plots indicate configuration size on the y-axis as $\log_2(WindowSize) - 5$ (e.g., *F-1024* is shown as $y = 5$, *F-32* is $y = 0$, etc.). *UpDown* represents the MLP heuristic without hysteresis, *UpOnly* includes hysteresis. The figure also plots the dynamic IPC of *F-128*, as an indicator of phase behavior.

In the `libquantum` benchmark, both *UpDown* and *UpOnly* work well, because the innermost loop of this benchmark is very simple. Between profiled loads, there is little difference in the number of subsequent independent misses. Therefore, when running `libquantum` (left), both

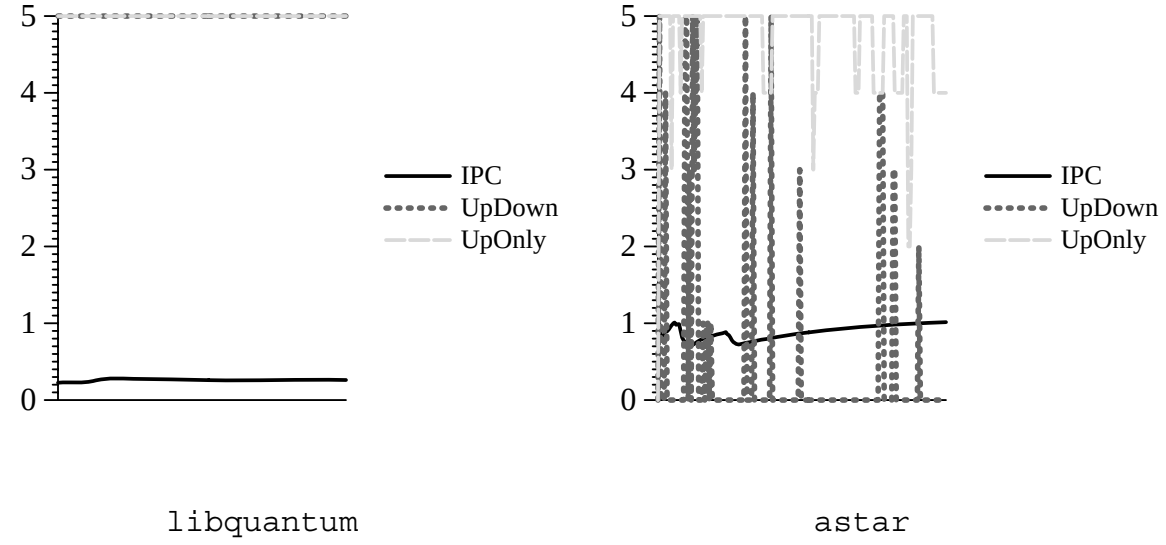


FIGURE 6-13. Configuration decisions over position, MLP heuristic with allowed down-scaling (*UpDown*) and disallowed down-scaling (*UpOnly*).

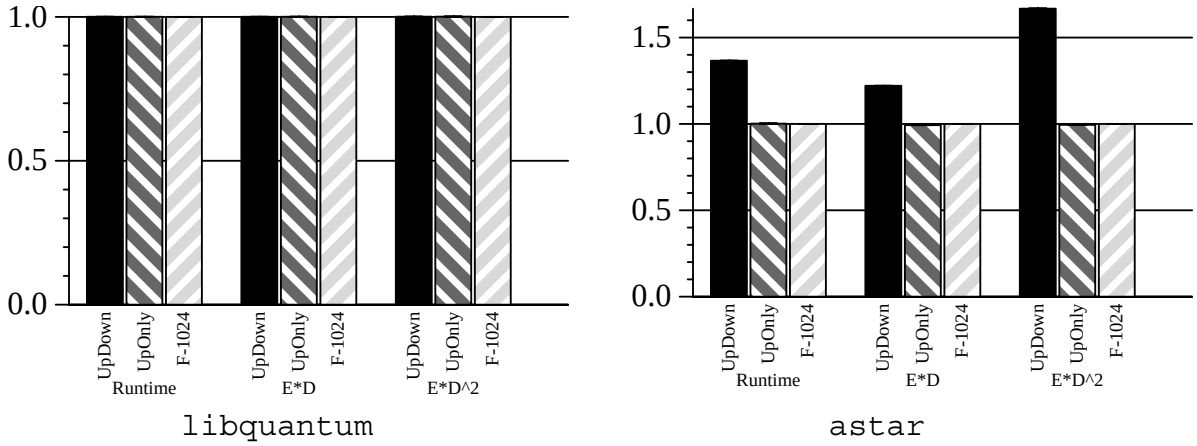


FIGURE 6-14. Normalized runtime, $E \cdot D$, and $E \cdot D^2$, MLP heuristic with allowed down-scaling (*UpDown*) and disallowed down-scaling (*UpOnly*), and best static configuration.

policies behave similarly (i.e., they choose to scale up completely, and remain fully scaled-up for the duration of the execution). On the other hand, `astar` *occasionally* exhibits an independent load miss exposed only by larger configurations. In `astar` (and the majority of benchmarks in this study), not all load misses are equal: some misses tend to be independent of others, and some misses are not. This leads to a varied estimation of MLP, as indicated by the large number of recon-

figuration decisions under *UpDown* in `astar`, as compared to *UpOnly*. Without hysteresis, this leads to causes of inefficiency. First, after scaling down, the processor is ill-equipped to expose MLP on subsequent load misses. Second, the act of reconfiguring requires quiescing the pipeline, incurring performance overhead at each decision point.

Figure 6-14 plots the resulting runtimes and efficiencies, normalized to that of *F-1024*. Overall, though *UpOnly* requires no more hardware than that of *UpDown*, the former delivers much better energy-efficiency in most cases. Therefore, in the remainder of this section, I evaluate only *UpOnly*, under the name *MLP*.

Limitations. Inaccuracies in estimation of optimal window size may arise for several reasons. First, the approach outlined above only considers true-path committed instructions. It does not account for prefetching effects of false-path operations. Second, the approach assumes that all W_{max} instructions within a profiling interval could have resided within the window of the largest possible configuration—this may not be possible if the profiled instruction interval includes a serializing event, or if branch mispredictions limit the number of in-flight instructions (very common on average, cf Chapter 5). Lastly, without a strong indicator for when to scale *down*, the overall effectiveness of MLP-aware scaling might be limited for scalable cores with fairly inefficient scale-up (as opposed to Forwardflow, used in this evaluation, which is often most efficient at its largest scaling points).

6.4.2 Heuristic Evaluation

This section evaluates which of the above metrics deliver the best overall energy-efficiency. I compare each approach against the most-efficient static configuration, on a per-benchmark basis. This baseline is fairly harsh—identifying the most-efficient configuration would require a com-

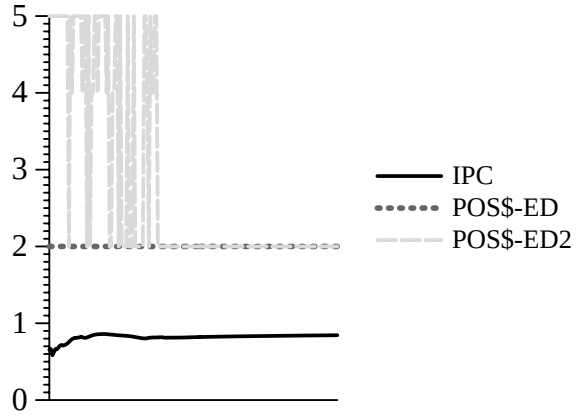


FIGURE 6-15. a) Scaling decision over position, *POS* optimizing $E \cdot D$ and $E \cdot D^2$, `leslie3d`.

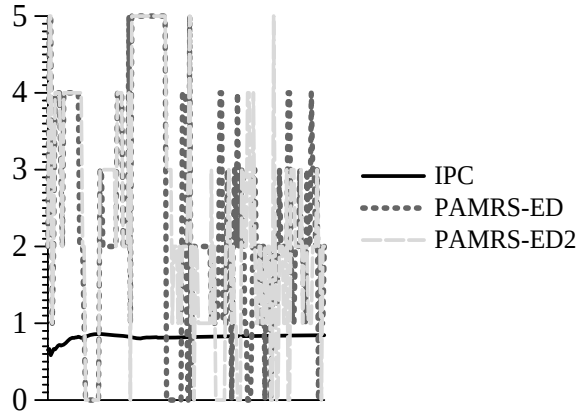


FIGURE 6-15. b) Scaling decision over position, *PAMRS* optimizing $E \cdot D$ and $E \cdot D^2$, `leslie3d`.

plete run of each configuration, in advance. Overall, all three techniques studied optimize efficiency metrics fairly well, close to or exceeding the efficiency of the baseline.

MLP does not seek to minimize any particular metric, but *POS* and *PAMRS* make measurement-based decisions. For *POS*, software analysis of the configuration space occasionally leads to different scaling decisions, depending on whether $E \cdot D$ or $E \cdot D^2$ is selected for optimization. This difference is indicated by two different *POS* configurations, *POS-ED* and *POS-ED2*. Figure 6-15 a) illustrates the only benchmark in which the scaling decisions differ significantly[7] under *POS-ED* and *POS-ED2* (`leslie3d`). Similarly, *PAMRS* is represented as two configurations, *PAMRS-ED* and *PAMRS-ED2*, depending on which metric is measured, though online sampling (i.e., *PAMRS*) yields significantly different scaling decisions in about 33% of examined benchmarks. Figure 6-15 b) exemplifies this behavior with a plot of `leslie3d`'s configurations in the course of execution.

7. My criteria for determining whether scaling decisions are "significantly different" is purely subjective, based on inspection.

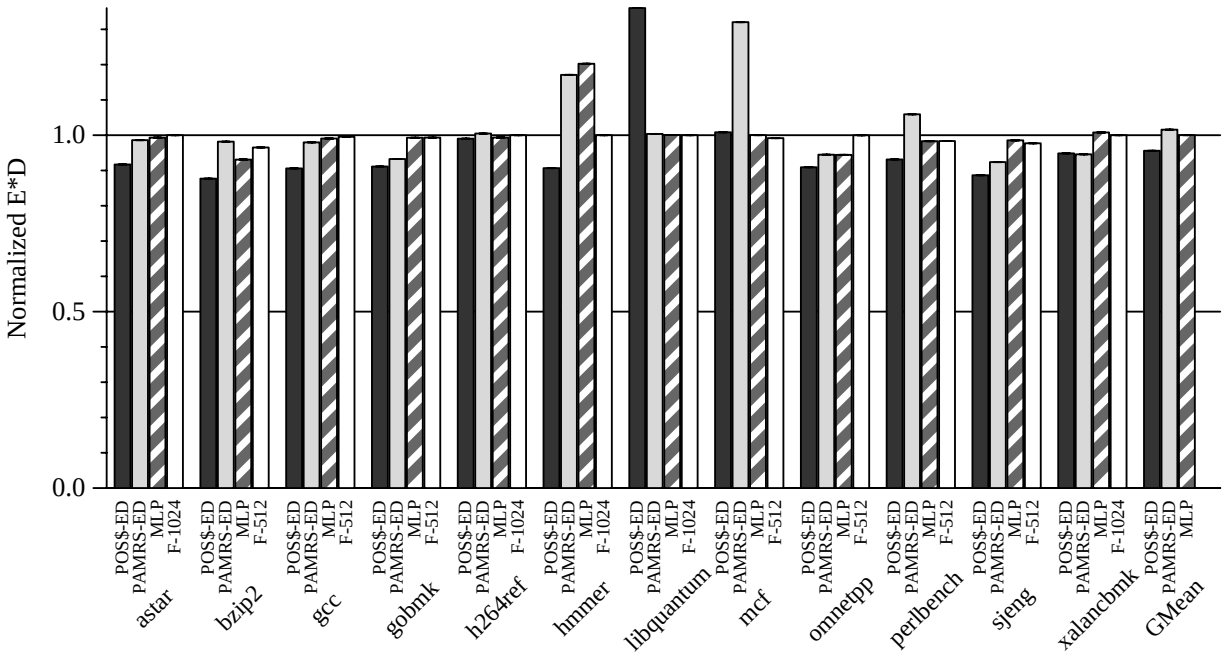


FIGURE 6-16. Normalized $E \cdot D$, SPEC INT 2006, dynamic scaling heuristics and best overall static configuration, normalized to *F-1024*.

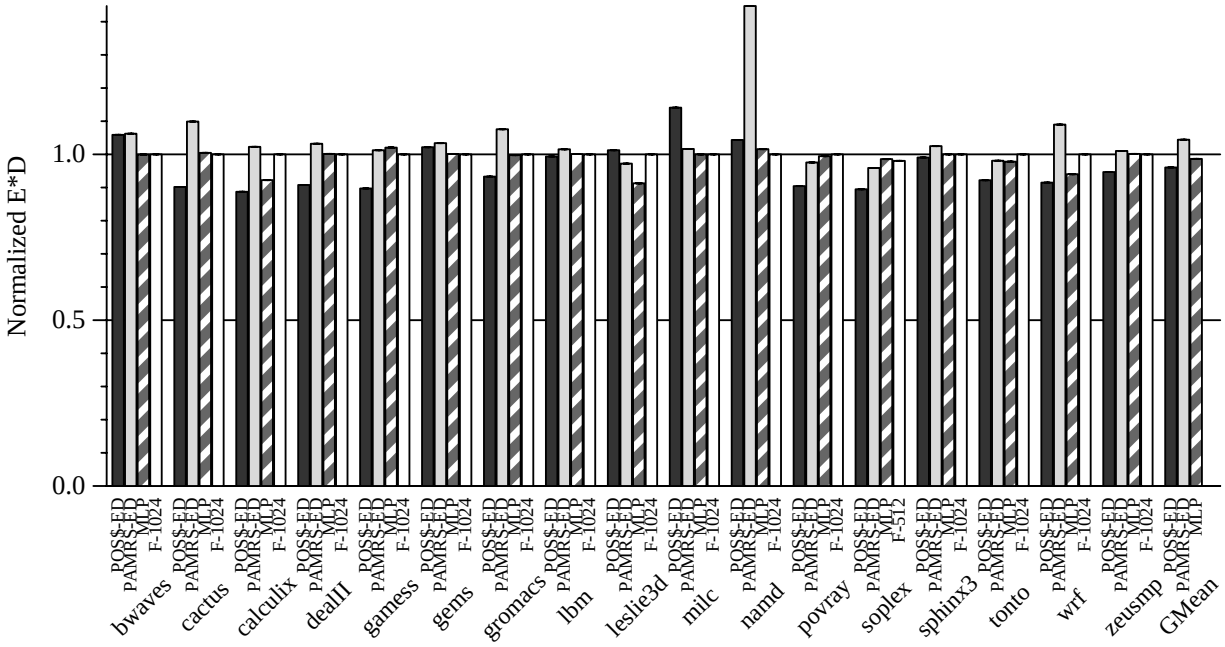


FIGURE 6-17. Normalized $E \cdot D$, SPEC FP 2006, dynamic scaling heuristics and best overall static configuration, normalized to *F-1024*.

In about a third of the examined benchmarks, *POS* and *PAMRS* yield different estimates for per-configuration efficiency, and therefore select different configurations (exemplified by

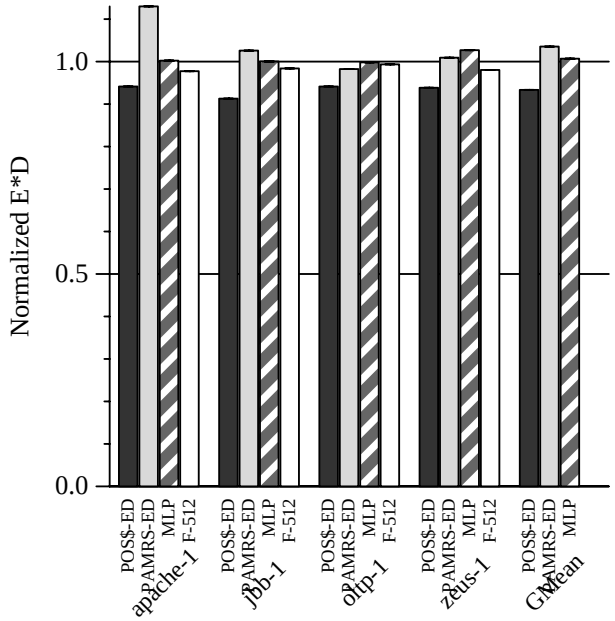


FIGURE 6-18. Normalized $E \cdot D$, Commercial Workloads, dynamic scaling heuristics and best static configuration, normalized to *F-1024*.

`leslie3d`). Since *POS* is usually very efficient (e.g., looking ahead to Figure 6-16), this trend suggests dynamic sampling does not adequately capture the true efficiency measures of each configuration: the wider-scoped static sampling of *POS* tends to do better. *POS*'s profiling captures the *overall efficiency* over a fairly large interval (e.g., a stack of function calls), whereas *PAMRS* captures a *sample* of the *local* efficiency. Though larger sampling intervals made no perceptible difference in *PAMRS*'s behavior in a sensitivity analysis, *POS*'s use of software-guided *indication* coupled with software-specified *decision* is usually more effective than *PAMRS*'s dynamic measurement approach.

Figures 6-16, 6-17, and 6-18, plot normalized $E \cdot D$ for integer, floating point, and commercial workloads, respectively, among all heuristics. The figures also include the static configuration which is most optimal for each benchmark. All efficiencies are normalized to that the of a single core fully scaled-up (i.e., *F-1024*). Overall, once the most-efficient static configuration is known,

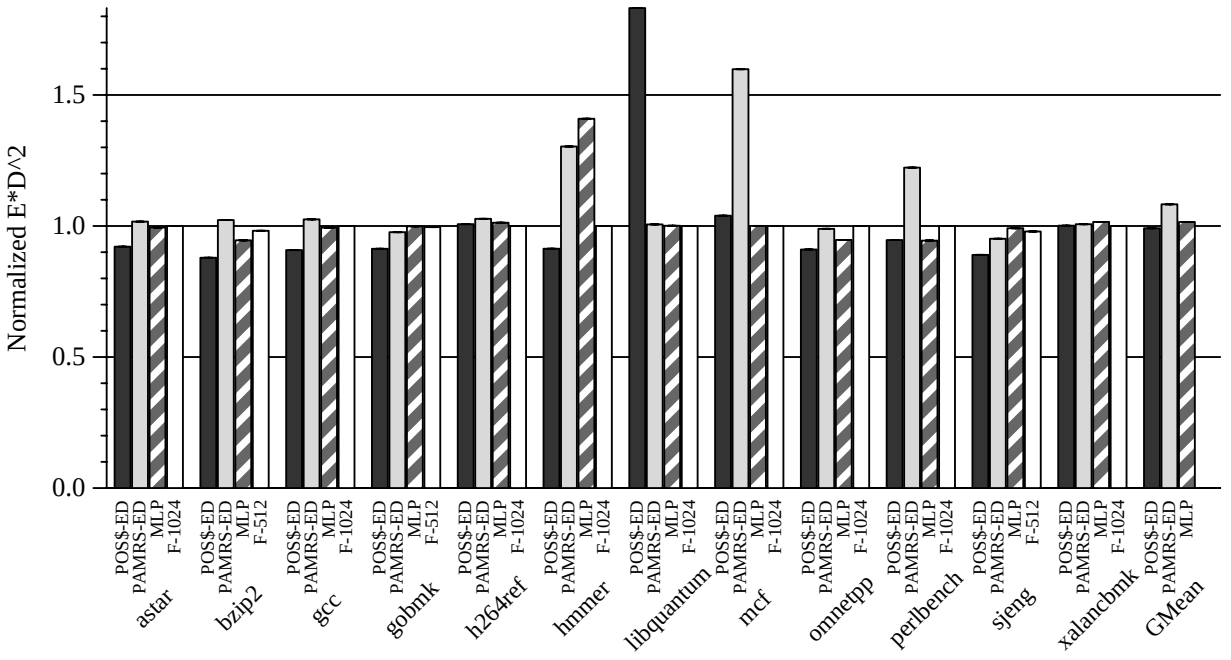


FIGURE 6-19. Normalized $E \cdot D^2$, SPEC INT 2006, dynamic scaling heuristics and best stati configuration, normalized to *F-1024*.

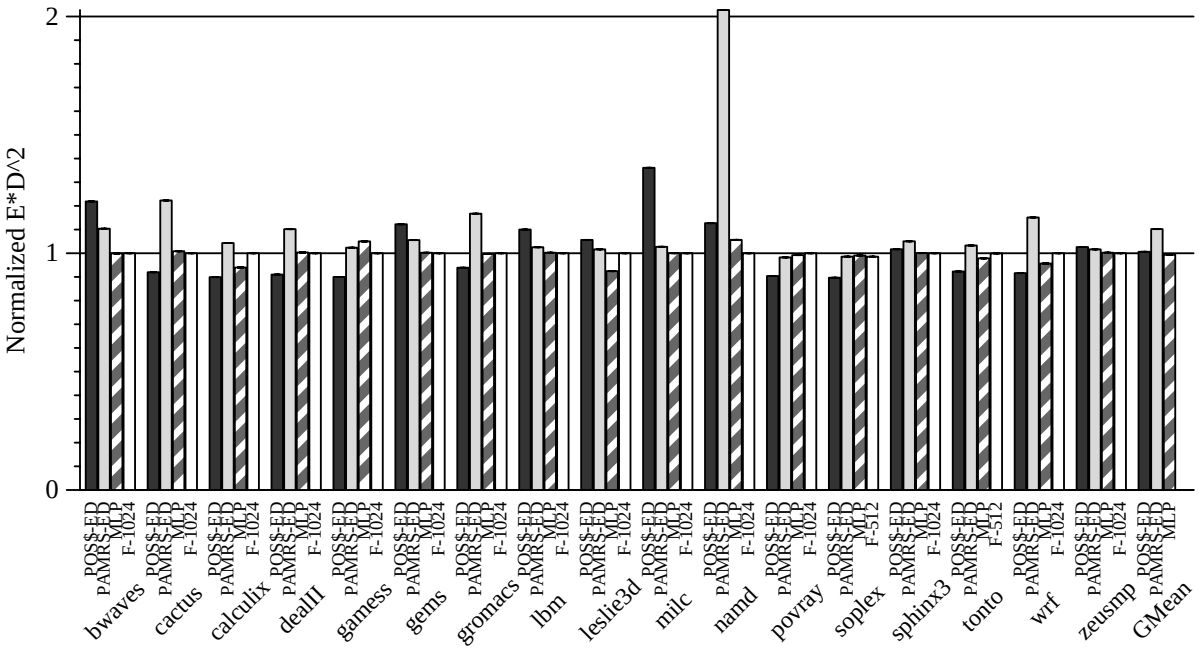


FIGURE 6-20. Normalized $E \cdot D^2$, SPEC FP 2006, dynamic scaling heuristics and best static configuration, normalized to *F-1024*.

fine-grain scaling for $E \cdot D$ shows a small average benefit under positional adaptation, essentially no benefit under *MLP*, and some reduction in efficiency under *PAMRS*. Trends are similar when

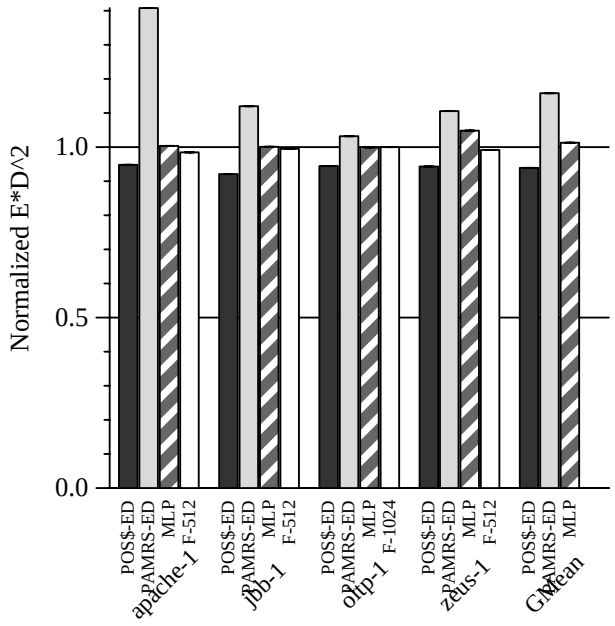


FIGURE 6-21. Normalized $E \cdot D^2$, Commercial Workloads, dynamic scaling heuristics and best static configuration, normalized to *F-1024*.

considering $E \cdot D^2$, in Figures 6-18 through 6-20.

Many benchmarks' efficiency improves somewhat under *POS*. *POS* delivers better efficiency than the best static configuration 67% of the time. This improvement exceeds 10% in 11 of 66 cases. Overall, POS implements the best *dynamic* scaling policy in 71% of all runs. However, the risk of significant degradation in efficiency should not be overlooked. The evaluation of *POS* used training inputs (where available) to ascertain the relative efficiencies—at least some of the benchmarks used in this study exhibit input-dependent behavior, which affects positional efficiency. *POS* reduces efficiency in excess of 10% in eight cases (e.g., `bwaves`, `milc`, and `libquantum`—benchmarks exhibiting better-than average window scaling, cf Chapter 5). Comparatively, *PAMRS* significantly reduces efficiency in 30% of cases, is only most efficient for `xalancbmk` optimizing $E \cdot D$ (Figure 6-16), and never significantly improves efficiency over the best static configuration.

Though *MLP* is seldom is more optimal than the most optimal static configuration (only 18% of the time), it can be used as a dynamic means by which to safely approximate the most-optimal static configuration, without an explicit profiling run. Moreover, it is less likely to harm efficiency, compared to *POS*, as *MLP* significantly degrades efficiency with respect to the best static configuration in only one benchmark (`hmmer`, an efficiency outlier for Forwardflow cores).

6.4.3 Fine-Grained Single-Thread Scaling Summary

When selecting a configuration to dynamically optimize energy efficiency, cores should occasionally scale down, when performance of scaled-up configurations does not justify the added power consumption. I have evaluated two prior proposals—Positional Adaptation [81] and Power-Aware Microarchitectural Resource Scaling [87], which attempt to directly optimize an energy-efficiency metric, through measurement and response. I have also proposed an MLP-based method by which to scale Forwardflow cores, which attempts to measure the microarchitectural events leading to efficiency, rather than the efficiency metrics themselves.

Though fine-grained scaling techniques do not often exceed the efficiency of the best static configuration, system software is not likely to know the best static configuration *a priori*. A hardware scaling policy can be leveraged by system software as a means to *identify* the best static configuration.

As a closing remark, because Forwardflow cores are used as an evaluation platform in these experiments, it is usually very energy-efficient to scale up. Most benchmarks are statically most-efficient under large windows. The generality of these findings may be subject to this effect, especially those of the *MLP* heuristic, which is tailored to the mechanism by which Forwardflow scales performance. However, a heuristic tailored to another scalable core would likely tend toward that

core's most-efficient operating point as well—in this sense, it is intended behavior. Such a policy plays to the strengths of the microarchitecture with which it is paired.

6.5 Scaling for Multi-Threaded Workloads

In this section, I evaluate means by which scalable CMPs can improve the energy-efficiency of multithreaded workloads. In general, a scalable core can positively affect a multithreaded workload's efficiency in two ways. First, by scaling up, sequential bottlenecks can be reduced in severity. In this area, the difficulty arises in identifying sequential bottlenecks in the first place. The second area of benefit is opportunistic scale-down: scaling down cores that perform no useful work (e.g., while waiting on a lock) conserves power at virtually no performance cost, thereby increasing efficiency.

A peculiarity of this work is that it seeks to *find* and *eliminate* sequential bottlenecks. In contrast, available multi-threaded benchmarks tend have been carefully organized to *avoid* sequential bottlenecks. Therefore, much of this evaluation focuses on a microbenchmark, which emulates a sequential bottleneck of varying severity, with a brief discussion other workloads.

6.5.1 Scalable Cores and Amdahl's Law

Amdahl's Law [12] (Eqn. 6.5) governs the reduction in runtime expected from parallel execution, as a function of the degree of parallelism, N, and the fraction of total work that can be executed in parallel, f.

$$RelRuntime(f, N) = (1 - f) + \frac{f}{N} \tag{6.5}$$

Equation 6.5 decomposes relative runtime into a sequential component (i.e., $1 - f$) and a parallel component (i.e., f, reduced in magnitude by a factor of N though parallel execution). However, Amdahl's Law implicitly assumes that all processing elements exhibit homogeneous performance. Hill and Marty [73] consider the effect of heterogeneous performance on Amdahl's Law, providing a basis for performance expectations of scalable cores (Eqn. 6.6).

$$RelRuntime(f, N, k) = \frac{1 - f}{k} + \frac{f}{N} \tag{6.6}$$

A CMP equipped with scalable cores still accelerates the parallel component of an execution by a factor N, but can also reduce the runtime of the sequential component of the calculation by a s*equential speedup factor,* k. Like f, the precise value of k varies by workload. In the context of a microarchitecturally-scaled core, k depends on how much ILP and MLP can be exposed by the scalable microarchitecture. In the specific case of the Forwardflow cores of this study, the sequential speedup factor is substantially higher for memory-bound workloads than for compute-bound workloads.

Figure 6-22 considers the potential runtime reductions possible under several different hypothetical *sequential work functions*, as well as the expected runtime predicted by Amdahl's Law without any core scaling. *SC, INT* considers a sequential phase consisting of a hypothetical workload that behaves similar to the SPEC INT 2006 benchmark suite ($k = 1.31$, when scaled from *F-128* to *F-1024*); *SC, FP* and *SC, COM* consider SPEC FP 2006 ($k = 1.53$) and commercial workloads ($k = 1.11$), respectively. Scalable cores help to optimize workloads with substantial sequential bottlenecks (i.e., smaller values of f): consequently, runtime reductions are more noticeable on the left-most data points.

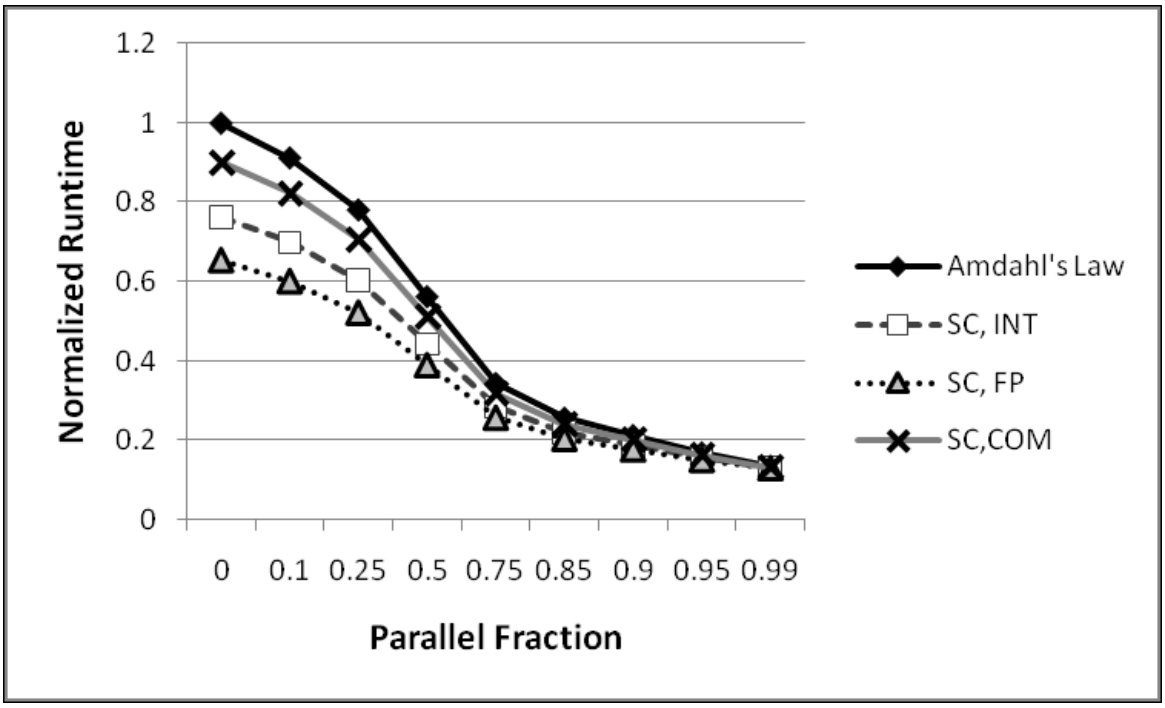


FIGURE 6-22. Predicted runtimes from Amdahl's Law (*N=8*), for static cores (Amdahl's Law), and scalable cores running integer (SC, INT), floating point (SC, FP), or commercial (SC, COM) workloads.

This section considers hardware policies for per-core scaling configurations, to achieve the hypothetical behavior depicted in Figure 6-22. In general, the challenge arises from difficulty in explicitly identifying parallel and sequential sections. Inability to ascertain whether a particular core's computation constitutes a sequential bottleneck or not hampers a scalable CMP's ability to ameliorate these bottlenecks through core scaling.

I discuss six approaches that seek to optimize energy-efficiency. First, under some circumstances, system software can be expected to adequately handle the problem of identifying sequential bottlenecks, directly achieving the behavior of Figure 6-22. Second, skilled parallel programmers know Amdahl's Law[8], and may themselves be aware of sequential bottlenecks. I

8. Though they may quickly forget it, according to Thomas Puzak.

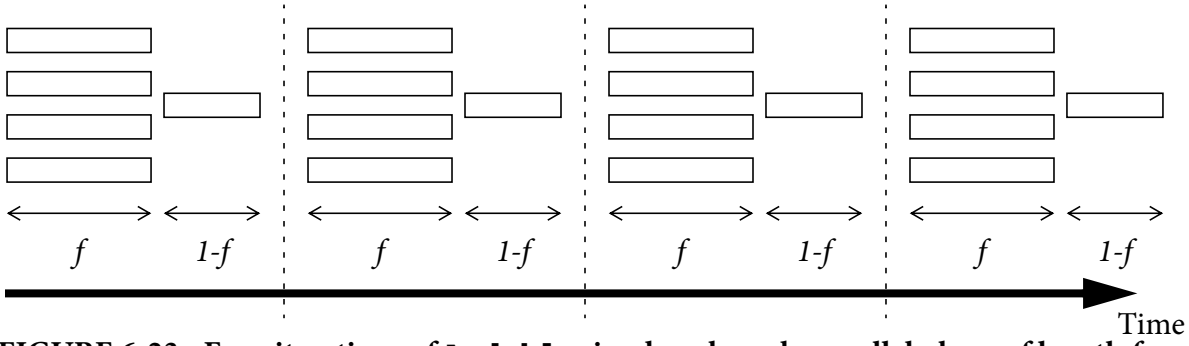


FIGURE 6-23. Four iterations of `Amdahl` microbenchmark, parallel phase of length *f*, sequential phase of length *1-f*.

evaluate programmer performance hints, over sequential bottlenecks of varying severity. Third, speculative lock elision (SLE) and related techniques ([134, 135]) use specialized hardware to identify critical sections. I consider the use of SLE-like critical section detection hardware as a means to identify a sequential bottleneck, and scale up accordingly. Fourth, hardware spin detection has been suggested a means to identify unimportant executions on multicore systems [182]. During spins, cores should scale down, not to improve overall performance, but to reduce energy consumption of effectively idle cores, thereby improving efficiency. Fifth, I consider using *both* spin detection to trigger scale down, and critical section identification to scale up. Last, I use spin detection as an indicator for *scale up* in non-spinning cores.

A Simple Microbenchmark. Few would argue that multithreaded benchmarks are simple. The implications of dynamically-scalable hardware on multithreaded benchmarks threatens to overwhelm intuition with a cross-product of complicated software and hardware behavior. To understand the effects of these scaling policies in a controlled environment, and to expose a sequential bottleneck (albeit artificially), I begin the evaluation with a microbenchmark, `Amdahl`. `Amdahl` consists of a simple computation kernel (randomized array traversal with a small computational payload), with alternating parallel and sequential sections of variable length. Literally, it executes

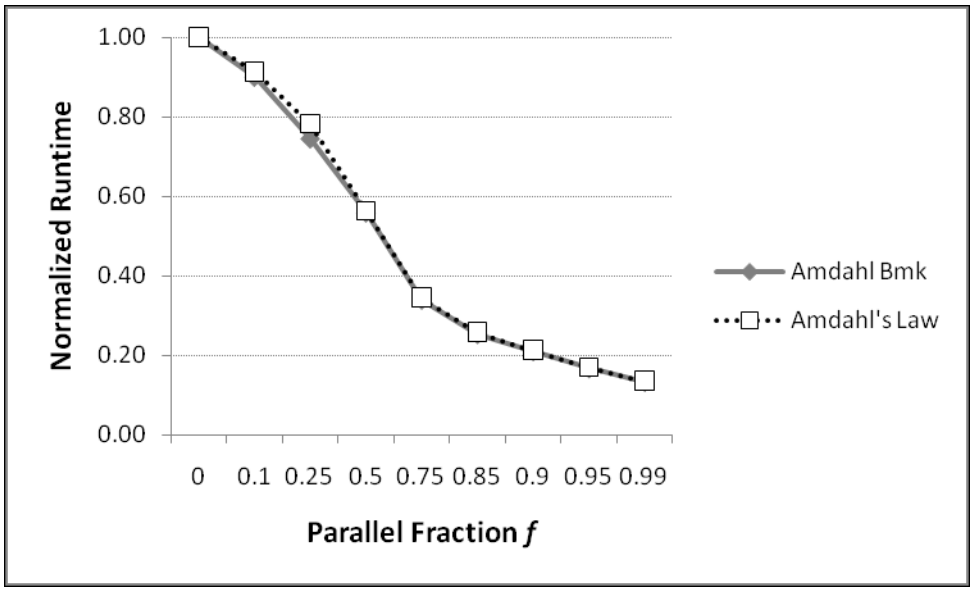


FIGURE 6-24. Runtime of `Amdahl` on 8-processor SunFire v880, normalized to $f = 0$, and runtime predicted by Amdahl's Law.

the process modeled by Amdahl's Law. Figure 6-23 illustrates four iterations of Amdahl, alternating between parallel and sequential phases. Figure 6-24 shows the normalized runtime of `Amdahl` as f is varied on an 8-processor SunFire v880 SMP, demonstrating the desired behavior of the microbenchmark. Simulations of `Amdahl` run for four transactions. `Amdahl`'s work function's runtime is reduced by about 40% when scaled from *F-128* to *F-1024* (i.e., $k \approx 1.65$, with respect to Eqn. 6.6).

6.5.2 Identifying Sequential Bottlenecks

System Software Scheduling and Scaling. When parallel and sequential phases are transparent to scheduling software, the behavior above is possible using system software to select per-core configurations. For instance, if a sequential bottleneck consists of a single *process*[9], upon which other *processes* explicitly wait, only the smallest degree of scheduling awareness is needed to accurately identify a sequential bottleneck. Under such a scenario, an operating system or hypervisor

can enact scaling decisions to directly achieve the desired behavior, even oblivious of f, N, and k. Given that this implementation should behave identically to the model of Eqn. 6.6 and Figure 6-22, I do not present a simulation-based evaluation of this scenario.

Programmer-Guided Scaling (*Prog*). It is not reasonable to expect a programmer to profile every phase of every application on all possible scalable core configurations. However, one might feasibly expect an expert programmer to use a combination of inspection, profiling, and intuition to manually identify sequential bottlenecks, parallel sections, and idle time in a parallel or concurrent application. To this end, I propose exposing the scaling features of a scalable core to the programmer, via a *performance hint instruction,* `sc_hint`.

```
while(true)
   if( tid == 3 ) {
      sc_hint(fast);
      doSequentialBottleneck();
   }
   sc_hint(slow);
   Barrier();
   sc_hint(default);
   doParallelSection();
}
```

FIGURE 6-25. `sc_hint` instruction example.

`sc_hint` communicates the programmer's performance preference to the scalable core. `sc_hint` takes a single argument, which indicates whether the core should scale up, e.g. to accelerate a sequential bottleneck (`sc_hint(fast)`), scale down, e.g., during a hot-spin or performance-insensitive section of the execution (`sc_hint(slow)`), or assume an OS-specified default configuration (`sc_hint(default)`), e.g., when executing a parallel section. Figure 6-25 shows how I annotate the `Amdahl` benchmark, described above, with `sc_hint` instructions.

9. I give emphasis to the process abstraction, because the scheduling state of processes are immediately available to system software, however, other means to communicate a sequential bottleneck are feasible, so long as the duration of the bottleneck is long enough to effectively amortize the cost of invoking the operating system.

Note that an implementation of `sc_hint` could take several forms. An actual instruction could be added to the ISA to accommodate a performance hint, or a control register (privileged or otherwise) could be added to the core, making `sc_hint` an alias for a control register write. I advocate the latter option, as system software should virtualize the performance hint during thread migration, so that performance hints are always associated with the workload *from which* they are given, rather the core *to which* they are given. Moreover, system software should be granted means by which to control programmer-guided hints, for instance by setting the actual configurations used by the various performance hints. For simplicity, I assume `sc_hint(slow)` scales a core down to the smallest configuration (*F-32*), `sc_hint(default)` scales the local core to *F-128*, and `sc_hint(fast)` scales the core up fully, to *F-1024*.

Unfortunately, programmer involvement not only requires additional software effort, but also makes this technique essentially useless to existing software, which contains no explicit hints. As such, I only evaluate programmer-guided scaling for microbenchmarks. The effort required to augment other benchmarks with performance hints does not seem justified, especially given that the parallel benchmarks used in this study are heavily optimized toward high values of f.

Critical Section Boost (*Crit*). *Speculative lock elision* (SLE) [134] identifies critical sections through use of hardware atomic operations (expected to be a lock acquisition), coupled with a temporally-silent store (expected to be a lock release). I propose using these same mechanisms to identify explicitly-synchronized sections of code, and scale up (to *F-1024*) whenever such a section is encountered. The core operates at its nominal configuration (*F-128*) when not executing a critical section. Intuitively, this accelerates lock-protected portions of the execution.

This technique can be deployed entirely in hardware, and therefore has the potential to affect existing software without modifications. However, the degree of success expected by this technique will depend on sequential bottlenecks being explicitly demarcated with synchronization directives. In particular, the microbenchmark based on Amdahl's Law does not include a lock-protected sequential bottleneck. This omission was accidental, but fortuitously serves to provide insight in at least two ways, discussed below.

Spin-Based Scale Down (*Spin*). Hardware spin detection has been suggested a means to help manage power in overprovisioned multicore systems [182], by identifying executions that perform no meaningful work. Software spins for a variety of reasons, e.g., genuine idle periods (in the operating system) and latency-sensitive synchronization. During such a spin, no meaningful work is performed. As such, I propose using spin detection to trigger per-core scale down operations. Under this policy, a core immediately scales down (to *F-32*) whenever a spin is detected. The core operates at its nominal scaling point otherwise (*F-128*). Intuitively, this scaling down causes the core to spin more slowly, consuming less power as it does so. If the spin is genuinely unimportant to overall workload performance, scaling down improves overall energy-efficiency of the execution.

Like critical-section boost, spin-based scale down can be deployed entirely in hardware, and is therefore applicable to software without explicit changes to do so. Of course, software that seldom or never spins will see no benefit from this heuristic.

Fast Critical Sections *and* Slow Spinning (*CSpin*). Critical-section boost and spin-based scale-down can also operate together. Under this hybrid approach, cores scale up when executing a

critical section, scale down when spinning, and operate at the nominal configuration for the remainder of the execution.

Importantly, when executing a critical section that *includes* a spin (e.g., a nested lock acquisition), cores *scale down*, pessimistically assuming that the lock will be held for some time. In a well-organized application, intuition suggests that threads should seldom spin within critical sections. However, evaluation reveals that false positives from the critical section logic (described below) can occur, and spin-based scale-down can sometimes correct otherwise pathological behavior.

All Other Threads Spin (*ASpin*). The spin-based model used per-thread spinning to motivate a local decision to scale down. However, more sophisticated policies are possible when spin indicators are shared between cores. For instance, by process of elimination, when only one core is *not* spinning, one might infer the remaining core is executing a sequential bottleneck. This is the precise syndrome that would arise from an operating system running a single-threaded application (even if that operating system is entirely unaware of scalable cores), or more interestingly, a multi-threaded benchmark with a significant sequential bottleneck.

Following this intuition, the *ASpin* scaling policy implements the same spin detectors as *Spin*, but also communicates spin behavior to other cores. Whenever a core detects that all other cores' spin detectors are asserted, the remaining (non-spinning) core scales up. The core immediately scales down to a nominal configuration (e.g., *F-128*) if another core stops spinning.

The method by which *ASpin* communicates spin information among cores is idealized in the evaluation. The precise implementation of the logic for *ASpin* presents several tradeoffs, both in mechanism itself and in the degree to which the mechanism is made visible to software. In one extreme, dedicated signals can be routed between all cores (or subsets of cores) to communicate a

single bit of information (spin state). This approach requires no virtualization, but only operates as expected during a sequential bottleneck among all *hardware* threads—in particular, it does not identify sequential bottlenecks among software threads (which may be descheduled). Under some circumstances, this may constitute desired or undesired behavior. Further, this technique only applies to *sequential* (1-thread) bottlenecks, not the more general case of *n*-thread bottlenecks.

On the other hand, this feature can be implemented mostly in software, albeit with greater expected overheads. For instance, if the spin detector is used to trigger a core-local exception upon activation, low-level software can maintain a count of spinning cores, or if desired, spinning threads. Software may react as desired to spins, e.g., with an inter-processor interrupt to a non-spinning core to force scale-up. Hybrids between all-hardware and mostly-software are also possible—the precise solution will depend on the degree of vertical integration enjoyed by a CMP manufacturer.

Lastly, *ASpin* may raise virtualization challenges for the system software. For instance, a software-visible spin indication line may require virtualization among threads within a cooperative process (e.g., within a multithreaded process, a virtual machine, or a Solaris-style processor set [161]). Under such virtualization, it would be beneficial to separate bottleneck detection (*indication*) from the scale-up response (*decision*), in the context of chip power budget shared at run-time between many multi-threaded applications.

6.5.3 Evaluation of Multi-thread Scaling Policies

Microbenchmark Evaluation. Figure 6-26 plots the normalized runtime of `Amdahl` for a variety of parallel fractions *f*, among all scaling policies; Figure 6-27 plots normalized power con-

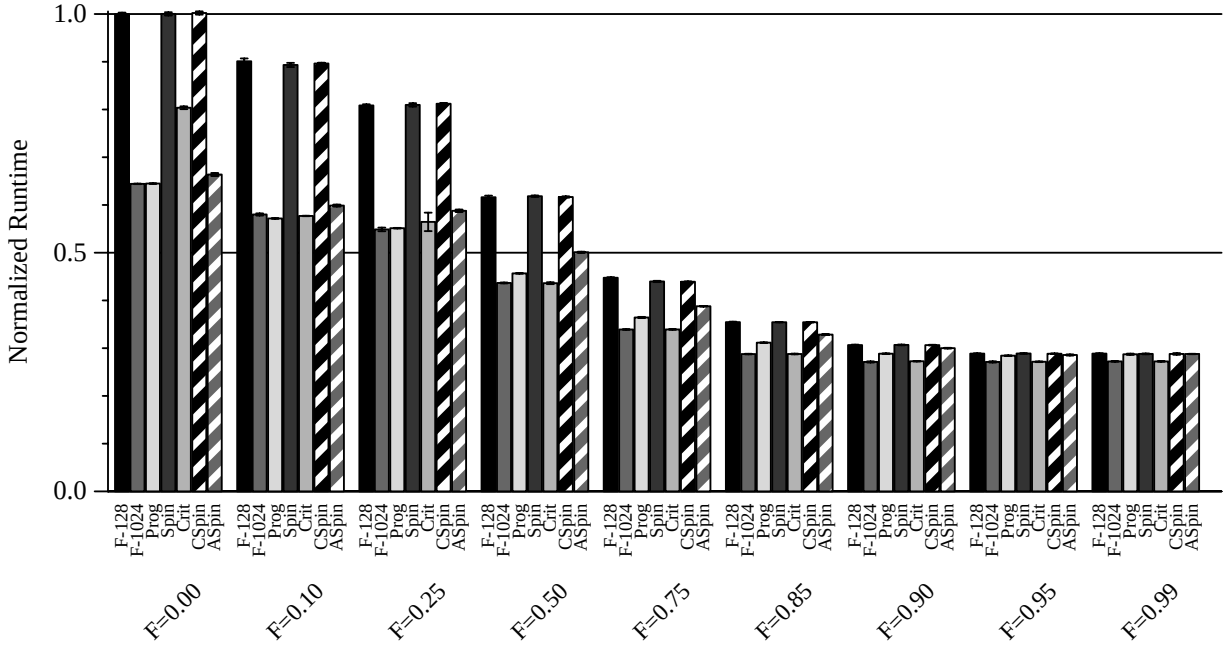


FIGURE 6-26. Runtime of `Amdahl` with varied parallel fraction *f*, all multithreaded heuristics (normalized to static configuration *F-128* for all cores).

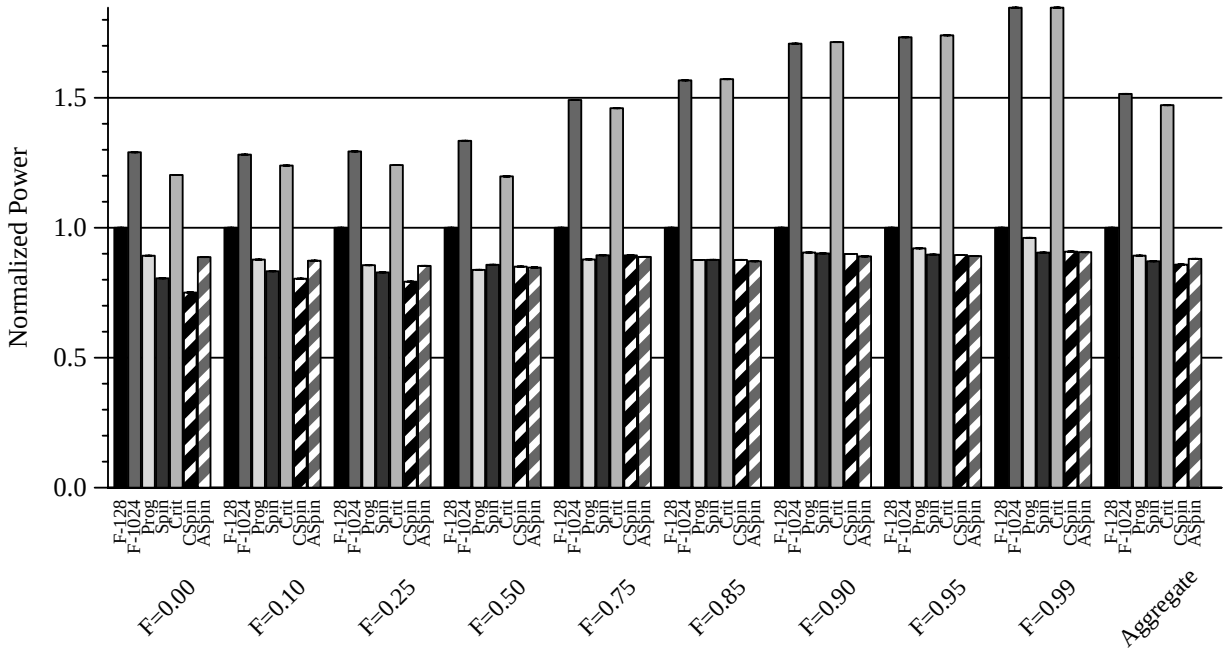


FIGURE 6-27. Power consumption of `Amdahl` with varied parallel fraction *f*, all multithreaded heuristics (normalized to static configuration *F-128* for all cores).

sumption. Runtimes are normalized to that of a static *F-128* configuration's runtime on $f = 0.0$. Figures 6-28 and 6-29 plot $E \cdot D$ and $E \cdot D^2$, respectively, normalized to *F-1024*. Overall, runtimes

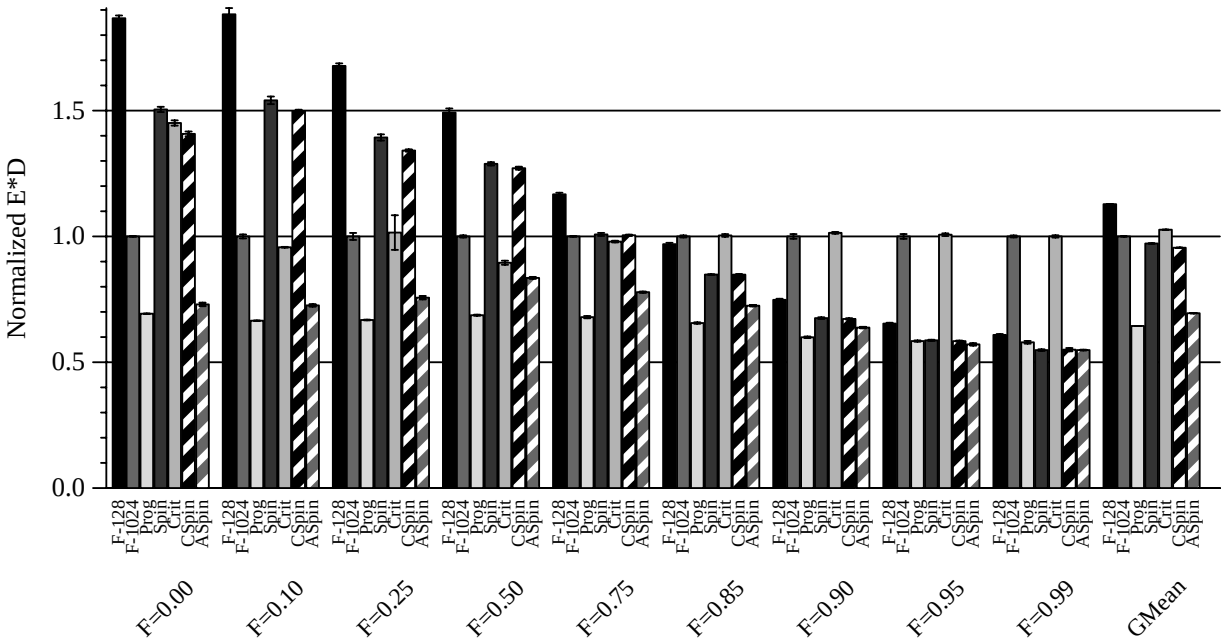


FIGURE 6-28. Efficiency ($E \cdot D$), normalized to *F-1024*, `Amdahl` microbenchmark, across all scaling heuristics.

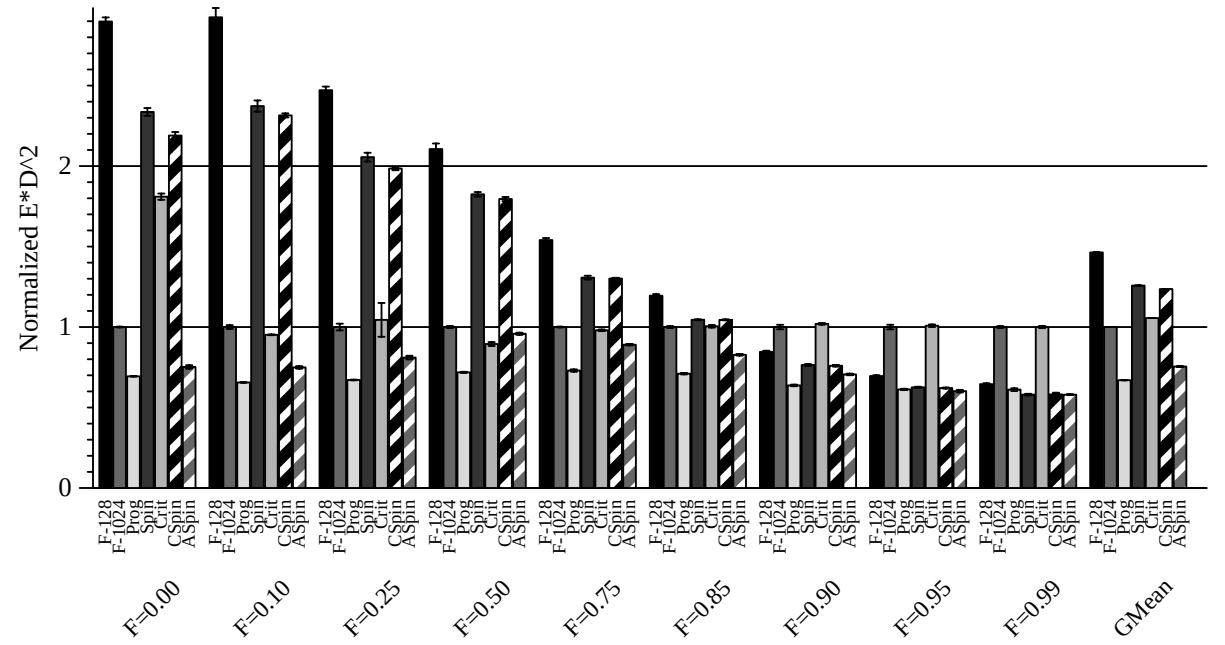


FIGURE 6-29. Efficiency ($E \cdot D^2$), normalized to *F-1024*, `Amdahl` microbenchmark, across all scaling heuristics.

follow (at worst) those predicted by Amdahl's Law (Eqn. 6.5), and at times, those of Amdahl's Law in the Multicore Era (Eqn. 6.6). Under *F-128* and larger configurations, `Amdahl` sees little or no

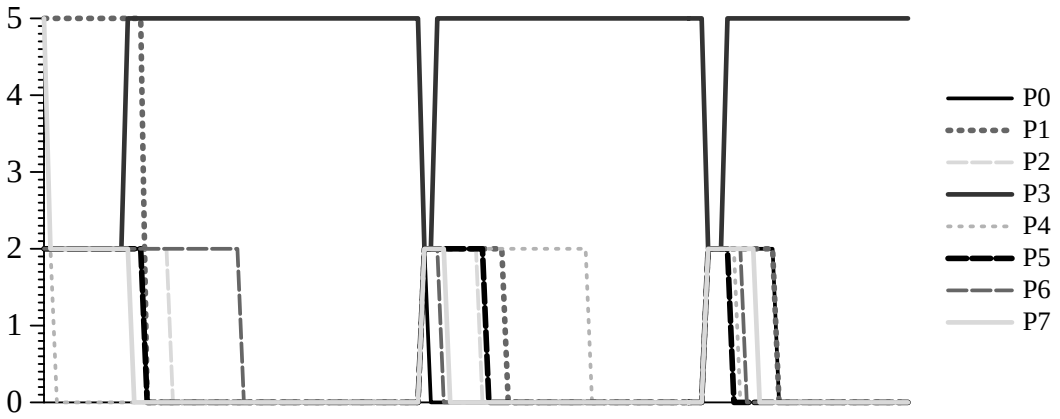


FIGURE 6-30. Scaling decisions over time, `Amdahl` $f = 0.85$, programmer-guided heuristic.

performance improvement beyond $f = 0.95$: this is an artifact of the small scale of the benchmark (intended to keep simulation times manageable). Under longer runs with larger working sets, further scaling is possible. In a related phenomenon, the performance difference between *F-128* and *F-1024* becomes less significant with increasing f. Greater explicit parallelism leads to greater load imbalance from individual high-performance cores, as synchronization becomes a bottleneck.

Programmer-guided scaling (*Prog*) is highly effective for all values of f. For low f, performance of the programmer-guided policy follows that of *F-1024*. As parallelism increases, performance tends toward *F-128*. Given sufficient programmer knowledge of how a parallel program operates, programmer-guided scaling is highly effective. Figure 6-30 plots the resulting changes in configuration over three transactions of `Amdahl`, $f = 0.85$, under the programmer-guided heuristic. Configurations in this and subsequent graphs are plotted on the y-axis as $\log_2(WindowSize) - 5$ (e.g., *F-1024* is shown as $y = 5$, *F-32* is $y = 0$, etc.). Even within the scope of a controlled microbenchmark, the effects of system-level activity are visible, e.g., the misalignment of parallel phases on cores. In this example, the sequential phase of the application is executed by

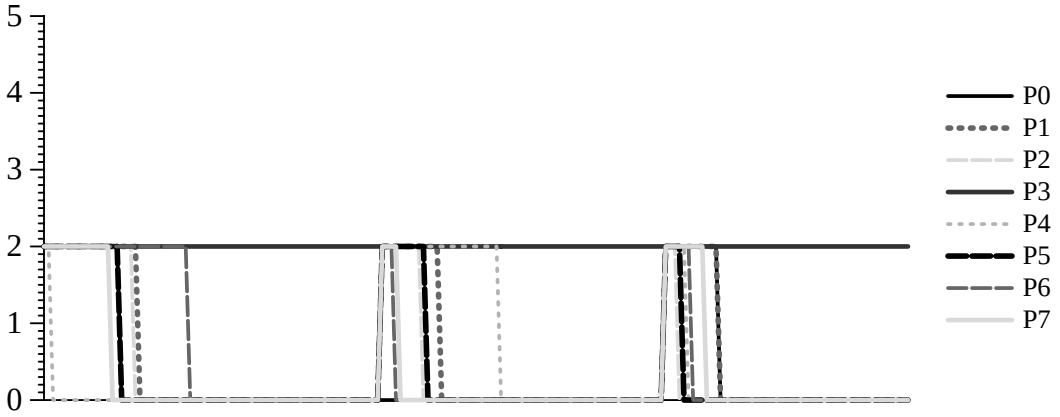


FIGURE 6-31. Scaling decisions over time, `Amdahl` $f = 0.85$, spin-based heuristic.

processor P3. In almost all cases, programmer-guided scaling is most efficient overall, and delivers best performance. Notably, this is only *not* the case when programmer expectations are not met by the execution (i.e., $f = 0.99$, in which the programmer "expected" performance to scale beyond that of $f = 0.95$).

Figure 6-31 plots the configuration space over time for the spin-based scale-down heuristic, *Spin*. Without an indicator for scale-up, cores never scale beyond *F-128* ($y = 2$). However, the spin detector correctly identifies cases in which processors wait for P3 to complete the sequential bottleneck, and scaled down accordingly, very similar to the programmer-inserted behavior of Figure 6-30.

Though *Spin* does nothing to decrease runtime, it noticeably improves efficiency, by reducing the power consumption of cores which perform no useful work. This effect is apparent across all sizes of sequential bottleneck, but is more noticeable at lower values of f.

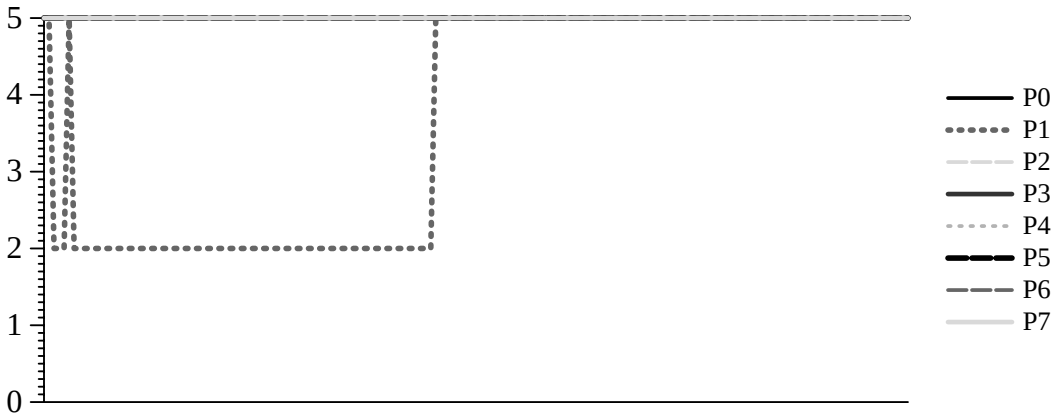


FIGURE 6-32. Scaling decisions over time, `Amdahl` $f = 0.85$, critical-section boost heuristic.

The scaling decisions of the critical section boost heuristic *Crit* are somewhat unexpected (Figure 6-32). Nearly all cores scale to the largest configuration (*F-1024*), and remain at that configuration for the duration of the benchmark. This is counterintuitive, as the critical section of Amdahl is not protected by a lock at all (refer to Figure 6-25). However, the implementation of barrier logic used in this microbenchmark *does* use a lock, which in turn triggers *false positives* in the implementation of SLE-style critical section detection logic. This represents an unintentional, but interesting, effect of the `Amdahl` benchmark: *its critical section is not lock-protected*. As a safety measure, the *Crit* automatically scales down after a fixed interval (1M instructions), but each invocation of the barrier triggers a new "critical section", resetting this counter. While the effect of critical section boost's false positives improves performance, like that of programmer-guided scaling, efficiency suffers from running spinning threads at large configurations.

Crit's false positives raise two important findings. First, *critical sections do not necessarily imply sequential bottlenecks*. `Amdahl` uses locks only to implement a barrier, which happens to give a false impression of criticality to the implementation of a critical-section detector used in this

study. Intuitively, however, it follows that other workloads will use locks for safety (not as a performance annotation), and *not all code sections guarded by locks will be bottlenecks*. Second, these false positives arise in the first place because the programmer's use of synchronization did not match the expectations of the hardware designer[10]. This is a cautionary tale, as *software* implements synchronization, and is free to do so in a variety of ways [112, 114], with or without locks [71], or even on an ad-hoc basis. A hardware mechanism to correctly detect all of these cases, including those not yet conceived, is intractable. False positives and their associated negative effects are likely unavoidable when scaling up based on perceived synchronization operations.

It may be possible to eliminate hardware inference of synchronization if synchronization libraries are explicitly augmented with the same annotations used in the programmer-guided-scaling experiment. A scalable-core-aware synchronization library could have detected `Amdahl`'s bottleneck, by observing all threads in a barrier, save one. Programs dynamically linked against a synchronization library would not even require a recompilation to leverage this functionality. Of course, this approach would not help applications using ad-hoc or lock-free synchronization, or those with statically-linked binaries.

Since *Crit* operates all cores at *F-1024*, it is not surprising that efficiency mostly follows that of the fully scaled-up configuration. The notable exception is $f = 0.0$, in which the "sequential" operation never encounters a barrier. Without the (beneficial) false positive effect, *Crit* never scales above a 128-entry window.

10. Ironically, *my* use of synchronization did not match *my own* expectations.

FIGURE 6-33. Scaling decisions over time, `Amdahl` $f = 0.85$, critical-section boost/spin-based scale-down heuristic.

The unexpected pathology in the critical section boost heuristic can sometimes be avoided when coupled with spin-based scale-down (i.e., *CSpin*). Figure 6-33 shows the scaling decisions from the hybrid spin/critical section scaling policy. Like the purely spin-based approach, most processors operate at size *F-128* or smaller for the duration of the execution. Processor P1 briefly scales up, again due to a false positive critical section (recall that P3 runs the sequential bottleneck), but is quickly checked by the spin detector. The efficiency of *CSpin* is comparable to that of *Spin*, as their behaviors are quite similar overall. Since *CSpin* effectively cures *Crit* of its pathology, I evaluate *CSpin* only in subsequent runs.

Figure 6-34 plots the scaling decisions of the *ASpin* policy, in which a thread (core) scales up only when all other threads' (cores') spin detector is asserted. Transitions demarcating iteration boundaries are very clear. In some cases, scheduling skew causes processor P3 to scale up shortly after it enters the sequential bottleneck (e.g., P3's scale-up is delayed by a slow P6 in the first iteration, and P4 in the second), but the decision is sufficiently timely to make an impact on perfor-

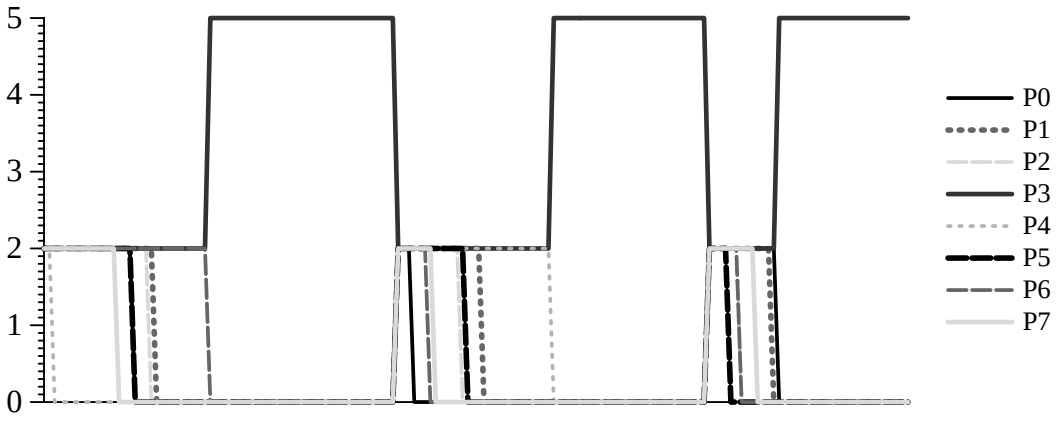


FIGURE 6-34. Scaling decisions over time, `Amdahl` $f = 0.85$, all-other-threads-spin heuristic.

mance (Figure 6-26) and efficiency (Figures 6-28 and 6-29). For `Amdahl`, *ASpin* effectively combines the beneficial scale-down of *Spin* and the intended critical scale-up of *Crit*.

Commercial Workloads. Using the insight from `Amdahl`, I now consider the behavior of commercial workloads under dynamic scaling heuristics. These benchmarks are quite different from the data-parallel microbenchmark: they are concurrent, task-parallel workloads. More to the point, they have been tuned to have few bottlenecks and large f, making them difficult candidates for dynamic scaling in general. *Prog* cannot be used on these workloads, hence its conspicuous absence in the figures belonging to this section.

The evaluation of the commercial workloads considers runtime per transaction as a measure of performance, plotted in Figure 6-35 for each of the four workloads. In general, heuristics focusing on scale-down (i.e., *Spin* and *ASpin*) do not significantly degrade performance, and they do improve energy-efficiency (Figures 6-36 and 6-37) with respect to a uniformly scaled-down configuration (*F-128*). Scaling up, however, is more efficient for `jbb`, provided power is available to do so.

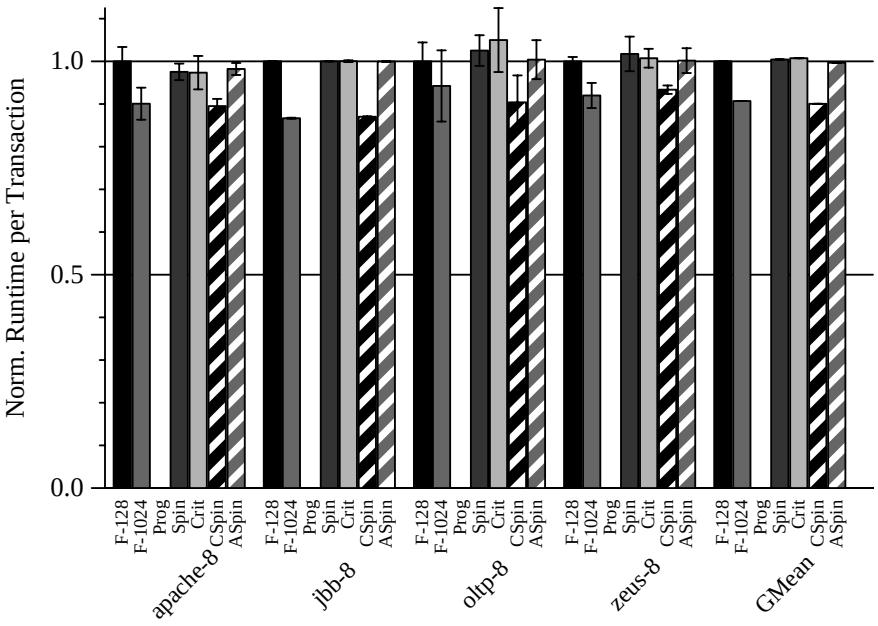


FIGURE 6-35. Normalized runtime per transaction, commercial workloads.

The commercial workloads are laden with synchronization—enough so that *CSpin*'s critical-section detection logic is very active. Due to the frequency of synchronization tempered with few spins, *CSpin* commonly causes all cores to scale up simultaneously. This may not be desirable for long periods, as it is not always energy-efficient to do so, and available power supply or cooling capacity may be exceeded by prolonged, multi-core scale-up. To illustrate, Figure 6-38 plots *CSpin*'s scaling decisions over time for the `oltp-8` workload. Other workloads exhibit similar behavior overall *CSpin*. `oltp-8`'s exact behavior is not sufficiently transparent to ascertain whether these critical sections constitute sequential bottlenecks or not: but runtime reductions are comparable to those of the static design point *F-1024*, suggesting that no specific bottleneck is remedied. As was the case with the `Amdahl` microbenchmark, it would seem that lock-protected critical sections do not necessarily imply sequential bottlenecks, at least in these workloads.

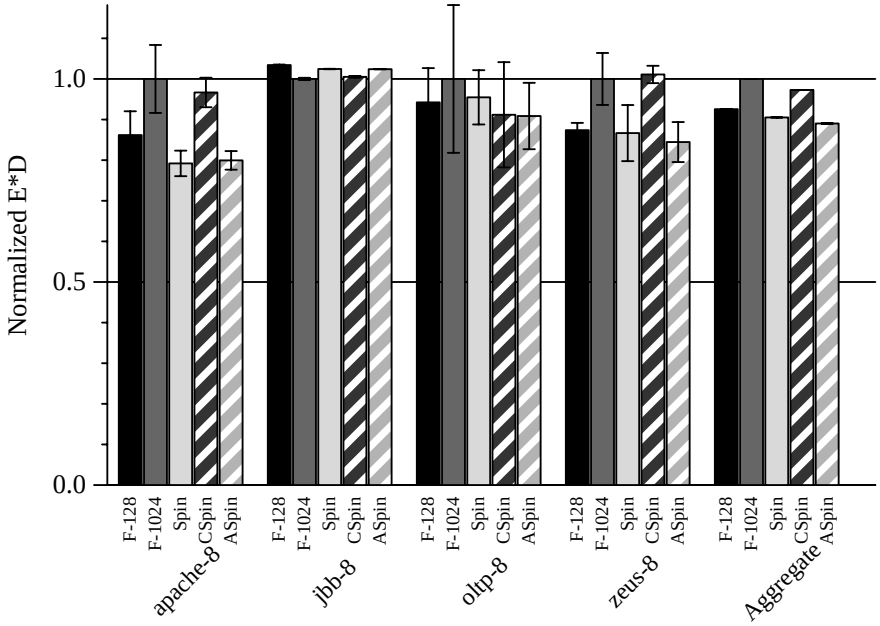


FIGURE 6-36. $E \cdot D$**, normalized to *F-1024*, Commercial Workloads.**

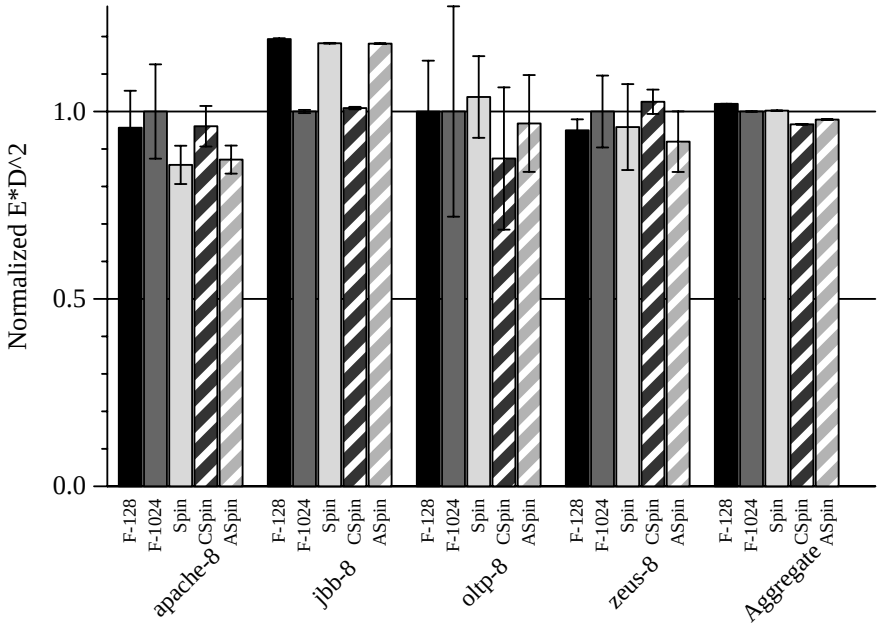


FIGURE 6-37. $E \cdot D^2$**, normalized to *F-1024*, Commercial Workloads.**

The spin-based approaches improve energy-efficiency by opportunistically scaling down. Figure 6-39 shows the scaling decisions of *ASpin* for `apache-8`, which shows the most frequent opportunistic down-scaling (though all workloads exhibit some). Importantly, despite the all-spin

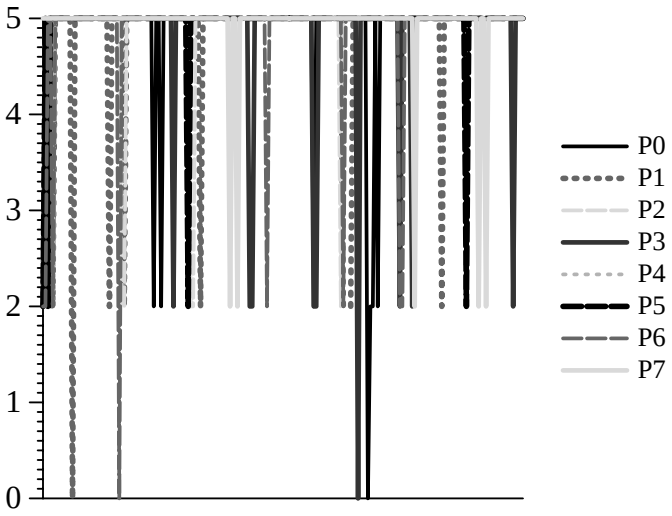


FIGURE 6-38. Scaling decisions over time, `oltp-8`, *CSpin* policy.

scale-up heuristic, *ASpin never scales up*. This suggests that in the `apache-8` workload, purely sequential bottlenecks are rare. This is not surprising, considering substantial effort was spent tuning the commercial workloads for scalability.

Summary. This section has shown that there is opportunity in fine-grained scaling for multi-threaded workloads, provided inefficient computations (e.g., spinning) or sequential bottlenecks can be identified and optimized appropriately through core scaling. For instance, programmer-guided scaling works well in a microbenchmark setting. Spin-detection hardware is effective at improving efficiency by opportunistically scaling down cores which perform no useful work. Lastly, though scaling up is often efficient by virtue of highly-efficient Forwardflow cores, using critical sections as an indicator to trigger scale-up yields counter-intuitive results: cores scale up when executing *critical sections*, not necessarily *sequential bottlenecks*.

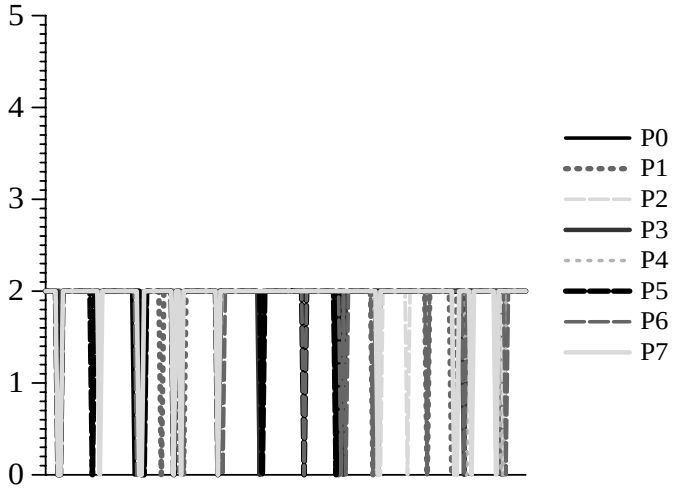


FIGURE 6-39. Scaling decisions over time, `apache-8`, *ASpin* policy.

6.6 DVFS as a Proxy for Microarchitectural Scaling

Regardless of whether scalable cores are coupled with hardware scaling policies, it seems likely that researchers will pursue software-level policies for dynamic management of scalable cores in CMPs. Section 6.3's findings indicate that software can use simple event-counting mechanisms to approximate power consumed by a single thread, and can use this and performance data to optimize a desired efficiency metric. But, development of these policies is challenging, as actual scalable hardware itself is not yet available for software evaluation.

One possible approach to this problem is to adopt the same methodology used in this thesis: full-system simulation. While this approach grants significant flexibility, e.g., in the types of scalable cores evaluated, the simple fact remains that simulation is extremely time-consuming. The single-thread evaluation used in this study executes only 100 million instructions per benchmark—i.e., the entire simulation covers about one typical scheduling time slice for today's operat-

ing systems. Nonetheless, such a simulation can require weeks of host execution time to complete. Barring major improvements in simulation latency and throughput, a meaningful evaluation via of multiple time slices would tax the resources—and the patience—of many systems researchers.

As an alternative to simulation, currently-deployed DVFS techniques seem a logical proxy for the behavior of future scalable cores. Though the future of DVFS-based scale-up is dubious (cf Chapter 2), DVFS-capable hardware is ubiquitous today. Altering frequency and voltage effectively varies a core's power/performance point, similar at a high level to the expected operation of a microarchitecturally-scaled core. However, DVFS speeds up a processor's operations; it does nothing to tolerate memory latency. This leads to performance disparities between the two techniques, of which future scalable core researchers should be made aware.

6.6.1 Estimating Window-Scaling Performance with DVFS

Forwardflow-style window scaling and DVFS approach performance scaling very differently. Window scaling improves performance by increasing core lookahead, which in turn enhances the core's ability to discover independent work to perform (i.e., instruction-level parallelism). In the best case, this results in the concurrent servicing of misses, increasing memory-level parallelism, and improving latency tolerance.

On the other hand, DVFS improves runtime by increasing frequency. Increasing frequency lowers clock cycle time, effectively reducing the absolute time required to perform operations *within* a core. That is, frequency-scaled operations require the same number of *cycles* to execute,

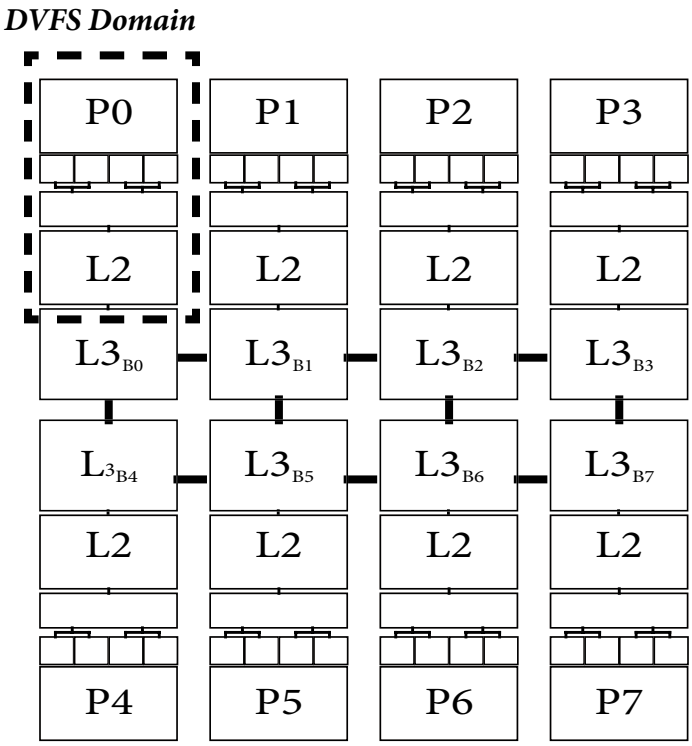


FIGURE 6-40. DVFS domain consisting of core and private cache hierarchy.

but the duration of a single cycle is lessened. Importantly, events outside the scaled frequency domain (e.g., a main memory miss) see no change in actual latency, and in fact may experience an *increase in perceived latency*, as measured in cycles.

Intuitively, these techniques attack different performance bottlenecks, and one would expect them to behave differently, depending on the requirements of a particular workload. A compute-bound application will benefit most from DVFS, as most of its execution time is devoted to computationally-intensive tasks. On the other hand, one would expect a memory-bound application to see little benefit from DVFS, and greater benefit from increased lookahead.

To explore this intuition, I present a simple experiment. With *F-128* as a baseline, Chapter 5 demonstrates that scaling to *F-256* (doubling the window size) can increase performance by about

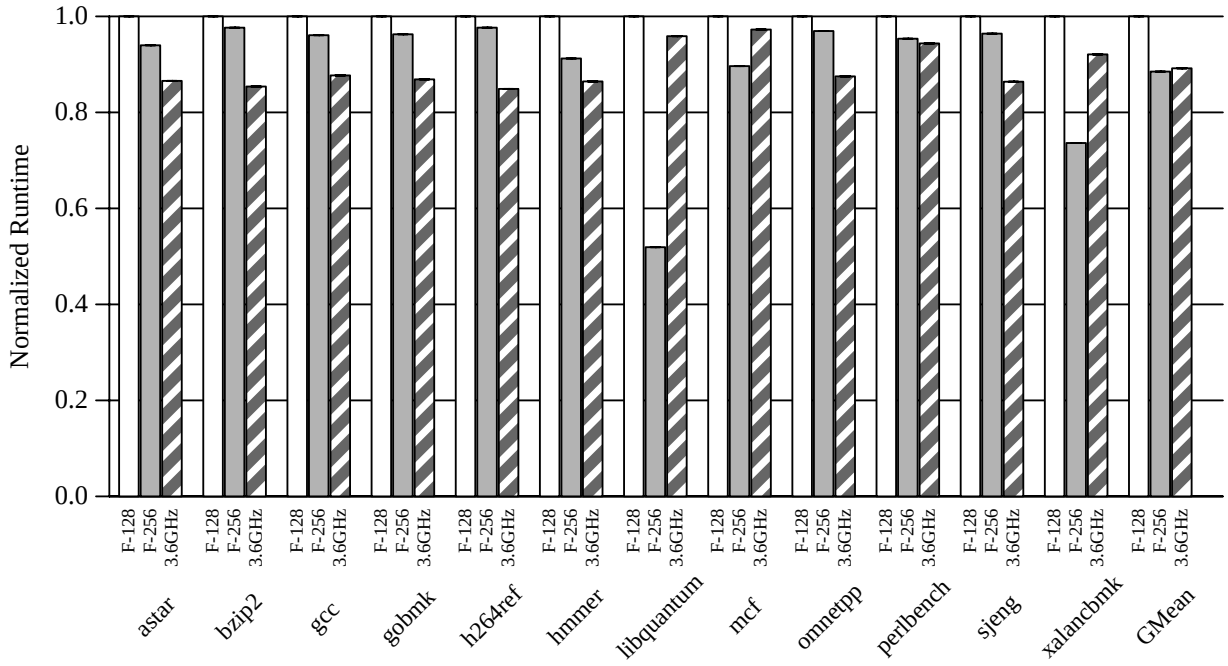


FIGURE 6-41. Normalized runtime, SPEC INT 2006, baseline Forwardflow core (*F-128*), partially-scaled Forwardflow core (*F-256*), and *F-128* accelerated by DVFS (*@3.6GHz*).

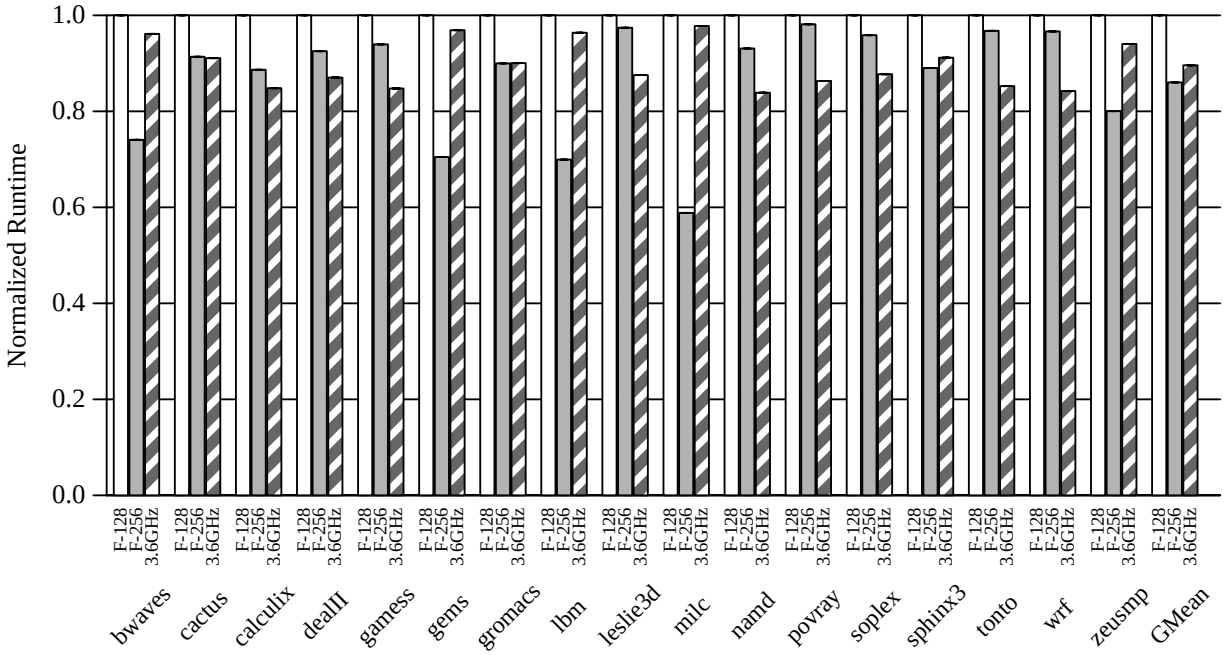


FIGURE 6-42. Normalized runtime, SPEC FP 2006, baseline Forwardflow core (*F-128*), partially-scaled Forwardflow core (*F-256*), and *F-128* accelerated by DVFS (*3.6GHz*).

10% for a single thread. DVFS can do the same. To achieve roughly the same speedup from DVFS, a 20% clock frequency increase is required[11]. I therefore consider two scale-up implementations in

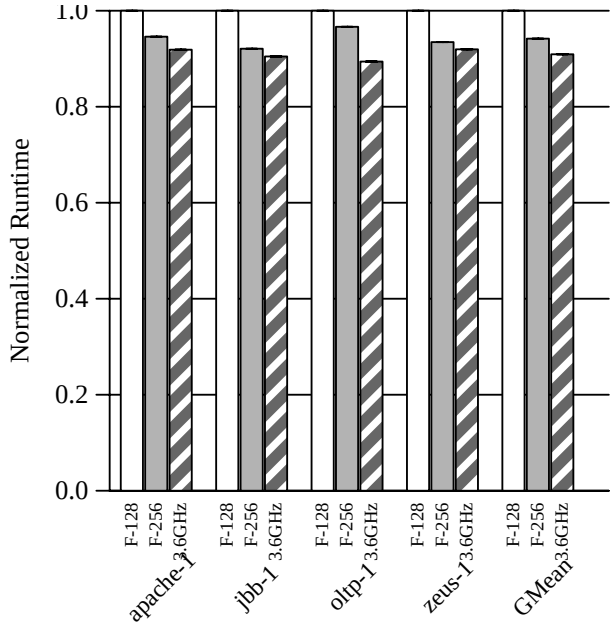


FIGURE 6-43. Normalized runtime, Commercial Workloads, baseline Forwardflow core (*F-128*), partially-scaled Forwardflow core (*F-256*), and *F-128* accelerated by DVFS (*3.6GHz*).

this section: microarchitectural scaling from *F-128* to *F-256*, and a +20% frequency increase on *F-128*'s clock, yielding the *3.6Ghz* configuration (cf Chapter 3, nominal frequency is 3.0 GHz). For experimental purposes, I assume each core in the 8-core CMP target machine can operate within its own DVFS domain: *3.6GHz* scales its *P0* core, L1, and L2 frequency and voltage up by 20%. The shared L3 remains at nominal operating conditions. I assume no delay for signals crossing asynchronous clock boundaries in this experiment. This assumption is optimistic for the DVFS case, but enables a per-core DVFS implementation without affecting the behavior (absolute timing or leakage) of the large on-chip L3 cache.

The comparison of *F-256* to *3.6Ghz* is fairly arbitrary. Both yield noticeably different runtimes and power consumptions than the baseline. But the most important aspects of this evaluation are

11. Determined empirically.

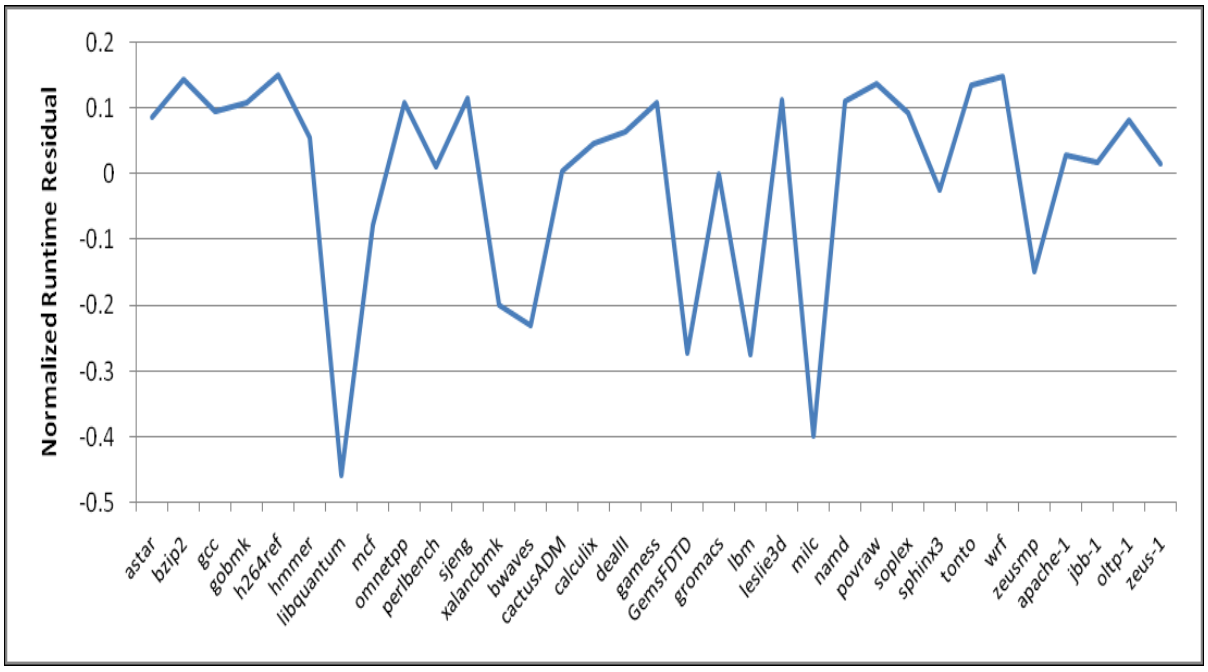


FIGURE 6-44. Relative runtime difference between DVFS-based scaling and window-based scaling.

qualitative in nature, and similar results are likely with other configuration pairs. E.g., an evaluation of *F-1024* and its average-performance-equivalent DVFS configuration would also suffice.

Figure 6-41 plots the normalized runtime of SPEC INT 2006, for the three configurations involved in this study. Normalized runtimes of SPEC FP 2006 and commercial workloads are plotted in Figures 6-42 and 6-43, respectively. Normalized runtime residuals between *F-256* and *3GHz* are plotted in Figure 6-44. In general, both DVFS and window scaling decrease runtime, as expected. However, there are two notable differences in performance behavior.

First, the degree to which each benchmark's runtime is reduced varies a great deal under microarchitectural scaling (standard deviation 0.12, 0.24 if *F-1024* is considered), from essentially zero (e.g., `sjeng`) to a roughly 2x speedup (`libquantum`). Runtime reductions from DVFS are typically much less varied (standard deviation 0.04).

Second, runtime reductions from microarchitectural scaling are negatively correlated with respect to those of DVFS (correlation -0.74). This finding supports the intuition that some workloads (i.e., compute-bound) benefit very little from microarchitectural scaling, but significantly from DVFS, and still other workloads (i.e., memory-bound) benefit little from DVFS but greatly from scaling.

Together, these two trends suggest that, among individual benchmarks, DVFS is a poor substitute for the performance characteristics of microarchitectural scaling. However, when averaged over many workloads, overall runtime reduction from DVFS is comparable to that expected of a scalable core.

In light of these findings, I recommend DVFS as a proxy for microarchitectural (window) scaling only for studies with rigorous evaluation of a meaningful variety of workloads. Such averages should include appropriately-weighted mixes of compute-bound and memory-bound workloads.

6.6.2 Estimating Window-Scaling Power with DVFS

Like performance, the effect of DVFS on overall power consumption is very different from that of microarchitectural scaling. To begin, consider the general equation for dynamic power consumption in CMOS:

$$P_{dynamic} = \alpha \cdot C_L \cdot f \cdot V_{dd}^2 \tag{6.7}$$

DVFS affects two terms in Eqn. 6.7. Frequency (f) is increased 20% in this experiment. If the baseline already operates at maximum possible (correct) frequency for nominal operating voltage, DVFS must also scale V_{dd} to ensure correct operation at greater frequency. Together, *scaling of*

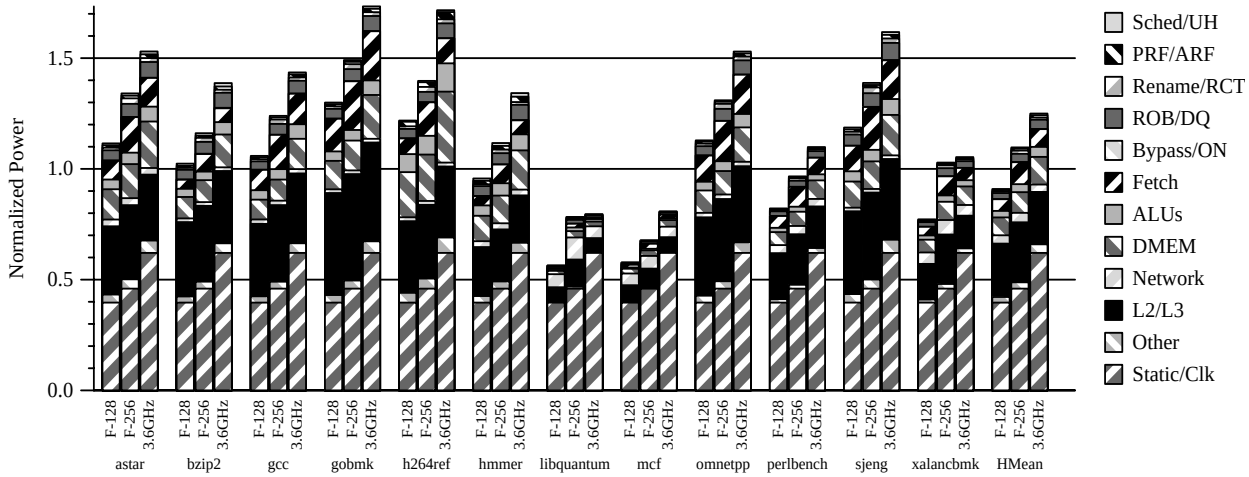


FIGURE 6-45. Normalized power consumption, SPEC INT 2006, baseline Forwardflow core (*F-128*), partially-scaled Forwardflow core (*F-256*), and *F-128* accelerated by DVFS (*3.6GHz*).

operating voltage and frequency leads to a cubic increase in dynamic power under DVFS. However, recall that the DVFS domain for this experiment is limited to a core and its private cache hierarchy. Only the power of these components will be affected by voltage scaling.

On the other hand, microarchitectural scaling holds frequency and voltage constant, and instead varies capacitive load C_L, by introducing additional resources when scaling up, as well as increasing activity factor α of unscaled components (recall that scaling window size also scales power of unscaled components, such as caches, by influencing the demands placed on those structures).

Voltage scaling's effects on static power are indirect. By affecting dynamic power, DVFS changes the expected operating temperature of nearby models. This in turn increases leakage power. There is also a small effect on the power of the clock distribution system (in fact, a dynamic effect) which is included in the 'static' category. By delivering clock pulses more frequently and at higher voltage, clock power increases cubically ($\alpha = 1$).

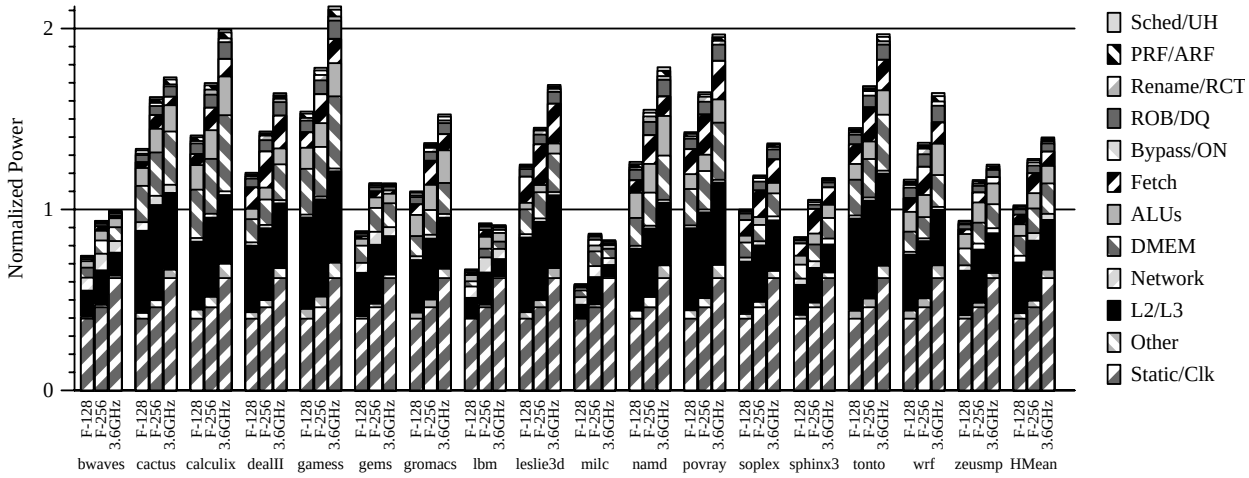


FIGURE 6-46. Normalized power consumption, SPEC FP 2006, baseline Forwardflow core (*F-128*), partially-scaled Forwardflow core (*F-256*), and *F-128* accelerated by DVFS (*3.6GHz*).

Figures 6-45, 6-46, and 6-47 plot categorized (normalized) power consumption across integer, floating point, and commercial workloads, respectively. A complete description of the categories can be found in Chapter 5, Section 5.4.1. The effect of DVFS and scaling on power consumption is immediately obvious. For comparable performance improvement, DVFS consumes substantially more power—both from dynamic and static sources. There are only two cases in which this trend does not hold (`lbm` and `milc`)—both of which exhibit significant performance improvements under microarchitectural scaling, accounting for atypically higher activity factors in *F-256* as a result.

Overall, DVFS *overestimates* power required to implement scale up, as compared to microarchitectural scaling. Researchers seeking to use DVFS to model scalable cores should do so with these caveats in mind. In particular, direct measurement of chip power changes from DVFS is not a reliable metric by which to gauge expected power consumption of a scaled-up core. Instead, a power model based on performance counters (Section 6.3) offers a reasonable alternative, as does (brief) simulation.

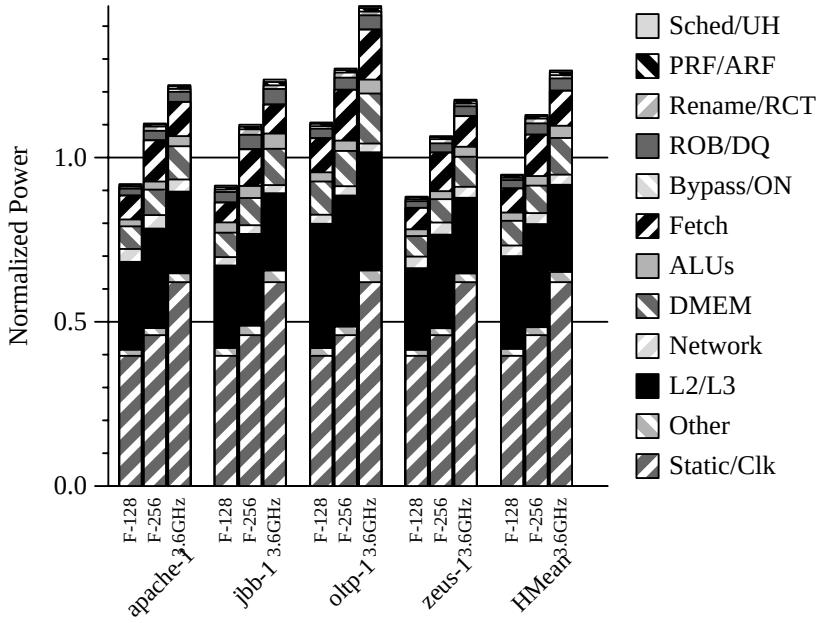


FIGURE 6-47. Normalized power consumption, Commercial Workloads, baseline Forwardflow core (*F-128*), partially-scaled Forwardflow core (*F-256*), and *F-128* accelerated by DVFS (*3.6GHz*).

6.7 Summary of Key Results

I have presented several key findings in this chapter. First, I have shown that per-core overprovisioning of scalable resources can be made cheap and simple, and further that it may enable "tighter loops" in core design [23]. While not all scalable core designs are well-suited to overprovisioning [86, 94], at least two core designs can used in the context of a scalable CMP with small cost to per-core overprovisioning [60, 180].

I have shown that scaling policies—both hardware- and software-based—can determine power consumption of single threads by computing linear weighted sums of core-local event counts. This computation can be performed directly at low cost, in hardware though *u*Op insertion, or in a software subroutine. Overall, the accuracy of this approximation technique matches

that of the methodology used in this thesis—accurate within 10-20% of true power consumption [25].

However, despite inexpensive methods for measuring power consumption, I find the most generally useful hardware scaling policy for fine-grained single-thread scaling does not directly measure power consumption. Instead, the proposed policy seeks to optimally size the instruction window by profiling available memory-level parallelism (MLP). I have shown that, by making temporally- or positionally-local efficiency measurements, some hardware scaling policies do not always adequately approximate global efficiency metrics. However, once the most-efficient static configuration is known (e.g., by profiling performed at the level of system software, possibly assisted by a hardware dynamic scaling policy), few observed single-thread benchmarks benefit further from dynamic scaling.

When multiple threads are considered, opportunistically scaling down to very low-performance and low-power configurations can improve efficiency. Candidate executions for scale-down can be identified by the programmer, or by spin-detection hardware. Among evaluated policies, only programmer-guided scaling reliably scales up when appropriate in a multithreaded setting—in particular, an evaluated implementation critical-section acceleration is ineffective and prone to false positives, leading to two findings. First, critical sections do always constitute sequential bottlenecks. Second, since synchronization is implemented in software, hardware-based methods of identifying synchronization, both free of false positives and accurate across all workloads, are likely infeasible, as programmers may employ difficult-to-detect or ad-hoc synchronization primitives at will.

Lastly, I have shown that future researchers seeking to explore software policies for scalable cores can look to DVFS as a method by which scalable core performance can be approximated, when considered in aggregate. Results based on individual benchmarks should not be considered significant, as the performance effects of microarchitectural scaling tend to vary more than those of DVFS. Researchers should not look to the power response of DVFS as an accurate model of scalable cores, however, as DVFS almost universally overestimates power consumption, due to its inherent cubic relation to dynamic consumption. Instead, researchers might consider simple event-based models for power consumption, like those evaluated in Section 6.3.

Chapter 7

Conclusions and Future Work

In this final chapter, I briefly summarize the contents of this thesis, and discuss future directions that could be investigated in further study of scalable cores.

7.1 Conclusions

This work makes several contributions toward the realization of a scalable CMP. First, I discuss SSR (Chapter 4), a dependence representation suitable for implementation in simple hardware. *SSR exploits the synergy between single-successor dominance and explicit successor representations*, by using pointers between instructions to explicitly define a dataflow graph. The principal finding of Chapter 4 is that the performance degradation of serialized wakeup of multiple successors can be masked, and overcome, by the increased the scheduling window size enabled by SSR.

Second, I discuss the architecture of an efficient, statically-scalable core (Forwardflow, Chapter 5). Intuitively, *in the shadow of a load miss, ample time is available to execute independent work*—Forwardflow exploits this intuition by leveraging SSR to build a large, modular instruction window, increasing lookahead instead of peak throughput. I thoroughly evaluate the Forwardflow design, and show that it outperforms similarly-sized conventional designs by avoiding scheduler clog, and that it can often exceed the performance of CFP and Runahead, while simultaneously using less power.

Next, I discuss a dynamically-scaled version of the Forwardflow architecture, and evaluate scaling policies for single- and multi-threaded workloads, to optimize energy-efficiency metrics. I show that multithreaded executions can be made more efficient by scaling cores down when they perform no useful work. Moreover, programmer-guided core scaling effectively identifies sequential bottlenecks, at fine granularity, and is the most effective evaluated method for deciding when to scale up when running many threads.

In the area of scalable core design, I show that *the area costs of fully-provisioned scalable cores can be small*, whereas *the potential performance loss from resource borrowing can be significant*. Future designers of scalable cores need not focus on whole-core aggregation to scale up, and can instead devote efforts to the (arguably easier) problem of implementing new, efficient overprovisioned scalable cores.

Lastly, I lay the groundwork for future research in software policies, by showing that DVFS can be used as a proxy for core scale-up, with some caveats. In particular, DVFS accurately models the performance of scalable cores, when averaged over many workloads. The rigor expected of a thorough scientific evaluation should provide this average. For modeling power, I show that simple correlation of microarchitectural events can effectively predict the energy consumption of a scalable core, suggesting that *future software policies can estimate the power consumption of scalable cores from performance counters, and the performance from today's DVFS*.

7.2 Areas of Possible Future Work

This thesis considers several areas of scalable core *hardware*: core microarchitecture, dynamic scaling policies, interconnects, etc., but significant work remains to be done in software. The experiments in this thesis have shown that software policies can estimate the power consumed by

single threads with high accuracy and low overhead. This motivates both software policies which are aware of per-configuration and per-benchmark power consumption, as well as future extensions to this work to consider multiple concurrent threads in a CMP.

I have shown that DVFS can be used to proxy performance of scalable cores, suggesting that future software-level considerations can use DVFS as a vehicle for evaluation, instead of simulation. Under DVFS, future policies for scalable CMPs can be discovered and evaluated well in advance of the hardware on which they will operate.

In the area of hardware, the benefits of energy-proportional design motivate future designs that push the extrema of scalable cores. More effective policies are possible when the underlying core grants greater flexibility in power/performance tradeoffs. Very high-performance/high-power or low-performance/low-power scalable core designs should be designed and evaluated. While SSR and the Forwardflow design may serve as baselines or even motivators for these future scalable core designs, there is no cause to believe that Forwardflow is the best overall scalable core design—it is merely the product of one thesis' work, and future designs are likely to surpass it, both in performance and efficiency.

This work has focused largely on cores, the effects of core scaling, and policies by which to scale cores. In particular, I have given little treatment to the memory system, how it should scale with core resources. Future architects should also consider schemes that aggregate cache space [86], effectively implement coherence [108], and manage cache space [22, 33].

7.3 Reflections

I conclude this document with a less formal consideration of the meta-lessons of this work. In this final section, I begin by placing this work in the context of the larger research community, and the trends underlying it. This section consists largely of my own opinions, which I do not attempt to substantiate quantitatively. These are my findings and impressions on research in computer architecture in general, rather than in the particular cases addressed in this document.

Much of the development of SSR and Forwardflow was motivated by the challenge to improve single-thread performance. Despite the final form of the presentation, only the latter half of the work leading to this dissertation had any notion of a scalable core—initially, I sought only to build an energy-efficient large-window machine. Ironically, scalable cores are ultimately motivated best in the context of emerging multi-threaded hardware. Before this work was adequately positioned in the context of future CMPs, it saw no acceptance among the research community.

This leads to my first main point: **core microarchitecture is at an impasse, because computer architecture is at an impasse**. ISCA 2010's panel, "Microarchitecture is Dead, Long Live Microarchitecture!" seemed to conclude with a consensus that microarchitecture is *not* dead, but in my opinion, no attendee pointed out convincing evidence of long life expectancy, either. In the looming shadow of commonplace highly-concurrent chips, I suspect not all researchers are fully convinced that the pursuit of single-thread performance is over. However, few would argue that the "low-hanging fruit" are gone.

One the other side of the impasse, it is not yet clear how software will effectively use concurrent hardware, beyond a handful of cores. Some might call this an opportunity: it constitutes a significant challenge for researchers, but somewhat discouragingly, four decades of research in

parallel computing has yet to solve the problem. On the other hand, with the entire computing industry now in the "Fundamental Turn Towards Concurrency" [162], motivation for the problem of parallel programming—and potential rewards for solving it—have never been higher. This work seeks to ease the transition into concurrent executions, by addressing sequential bottlenecks. If core microarchitecture has a future in academic computer architecture research, it is in the need to address sequential bottlenecks with new and novel approaches.

This leads to my second main point: **software must change to accommodate concurrency, and that trend opens the door for architecture to change as well** (e.g., in the ISA). In one of the successful proposals in this work, exposing core scaling to the programmer enabled a hardware scaling policy to partially ameliorate a sequential bottleneck—thereby (theoretically) helping the programmer meet his/her performance objectives. Exposing core scaling constitutes a straightforward tool by which the programmer can relate performance constraints to the underlying hardware and/or system software. Of course, much future work should evaluate the technical, linguistic, and psychological implications of performance hints (after all, there is a worry that programmers will compulsively litter their code with "go fast" directives, obviating the point of scalable cores). Certainly, the new primitives and abstractions that emerge as huge numbers of programmers tackle the multicore programming problem will shape expectations for years to come.

This does not obviate the need for faster cores. Faster cores will still be in demand for many years. On numerous occasions I desired a faster core—indeed, a scalable core—when developing the simulation software used in this study. This segues to the final meta-point of my discussion: I believe **researchers should seek new means by which to evaluate academic research**. Simulation

is hard, slow, error-prone, and quite frankly, it is difficult to convince oneself (or others) of its accuracy. Though it has been a staple of many academic (and industrial) research teams, and my own research, all of these problems' complexity grows worse as focus shifts to multicore designs. Ultimately, working with a simulator (even of my own design) *constrained* rather than *empowered* my research, by encouraging me to operate within its capabilities, rather than expand them. To quote Abraham Maslow's *law of the instrument*, "if all you have is a hammer, everything looks like a nail"[1]. With simulators as hammers, I believe there is a tendency to cast architecture problems as nails, i.e., to frame problems within the bounds of what can be easily simulated, rather than pursue the best means by which to make the evaluation.

I do not make this point lightly. A great deal of the technical effort leading to this thesis was spent in simulator development—which was an excellent software engineering experience—but gains nothing "on paper" as far as the research community is concerned. I find this curious. Methods affect the entire field, and it should be considered a first-order problem.

Unfortunately, I have no specific alternatives to offer on this point. Few general evaluation methods approach the near-panacea of simulation. Achieving the visibility, flexibility, and turnaround time of simulation with another methodology is daunting. Simulation may be a necessary evil in some cases, possibly as a means by which to bootstrap other forms of evaluation. Nonetheless, I believe the community would benefit from less simulation, on the whole. As computer architecture research takes the fundamental turn, an opportunity exists to change architecture in ways not previously possible, and at the same time, change the means by which we evaluate it.

---

1. Abraham H. Maslow. *Psychology of Science: A Reconnaissance.* Maurice Basset, Publisher. 1966 and 2002.

References

[1] I. Ahn, N. Goundling, J. Sampson, G. Venkatesh, M. Taylor, and S. Swanson. Scaling the Utilization Wall: The Case for Massively Heterogeneous Multiprocessors. Technical Report CS2009-0947, University of California-San Diego, 2009.

[2] H. H. Aiken and G. M. Hopper. The Automatic Sequence Controlled Calculator. *Electrical Engineering*, 65(8), 1945.

[3] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint Processing and Recovery: An Efficient, Scalable Alternative to Reorder Buffers. *IEEE Micro*, 23(6), Nov/Dec 2003.

[4] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proc. of the 36th Annual IEEE/ACM International Symp. on Microarchitecture*, Dec. 2003.

[5] A. R. Alameldeen, C. J. Mauer, M. Xu, P. J. Harper, M. M. K. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. Evaluating Non-deterministic Multi-threaded Commercial Workloads. In *Proc. of the 5th Workshop on Computer Architecture Evaluation Using Commercial Workloads*, pages 30–38, Feb. 2002.

[6] A. R. Alameldeen and D. A. Wood. Variability in Architectural Simulations of Multi-threaded Workloads. In *Proc. of the 9th IEEE Symp. on High-Performance Computer Architecture*, pages 7–18, Feb. 2003.

[7] A. R. Alameldeen and D. A. Wood. IPC Considered Harmful for Multiprocessor Workloads. *IEEE Micro*, 26(4):8–17, Jul/Aug 2006.

[8] D. Albonesi, R., Balasubramonian, S. Dropsbo, S. Dwarkadas, F. Friedman, M. Huang, V. Kursun, G. Magklis, M. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P. Cook, and S. Schuster. Dynamically tuning processor resources with adaptive processing. *IEEE Computer*, 36(2):49–58, Dec. 2003.

[9] D. H. Albonesi. Selective cache ways: on-demand cache resource allocation. In *Proc. of the 32nd Annual IEEE/ACM International Symp. on Microarchitecture*, pages 248–259, Nov. 1999.

[10] M. D. Allen, S. Sridharan, and G. S. Sohi. Serialization sets: a dynamic dependence-based parallel execution model. In *Proc. of the 16th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 85–96, Feb. 2009.

[11] R. S. Alqrainy. Intel Microprocessor History. http://www.scribd.com/doc/17115029/Intel-Microprocessor-History.

[12] G. M. Amdahl. Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities. In *AFIPS Conference Proceedings*, pages 483–485, Apr. 1967.

[13] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *Proc. of the 11th IEEE Symp. on High-Performance Computer Architecture*, Feb. 2005.

[14] K. Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, pages 300–318, Mar. 1990.

[15] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report Technical Report No. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[16] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. Jones, and B. Parady. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In *Workshop on OpenMP Applications and Tools*, pages 1–10, July 2001.

[17] R. Bahar and S. Manne. Power and Energy Reduction Via Pipeline Balancing. In *Proc. of the 28th Annual Intnl. Symp. on Computer Architecture*, July 2001.

[18] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proc. of the 33rd Annual IEEE/ACM International Symp. on Microarchitecture*, pages 245–257, Dec. 2000.

[19] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, S. Dwarkadas, and H. Dwarkadas. Dynamic Memory Hierarchy Performance Optimization. In *Workshop on Solving the Memory Wall Problem*, June 2000.

[20] A. Baniasadi and A. Moshovos. Instruction distribution heuristics for quad-cluster, dynamically-scheduled, superscalar processors. In *Proc. of the 27th Annual Intnl. Symp. on Computer Architecture*, June 2000.

[21] L. A. Barroso, J. Dean, and U. Holze. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*, 23(2), March 2003.

[22] B. M. Beckmann, M. R. Marty, and D. A. Wood. ASR: Adaptive Selective Replication for CMP Caches. In *Proc. of the 39th Annual IEEE/ACM International Symp. on Microarchitecture*, Dec. 2006.

[23] E. Borch, E. Tune, S. Manne, and J. Emer. Loose Loops Sink Chips. In *Proc. of the 8th IEEE Symp. on High-Performance Computer Architecture*, Feb. 2002.

[24] S. E. Breach. *Design and Evaluation of a Multiscalar Processor*. PhD thesis, Computer Sciences Department, University of Wisconsin–Madison, Feb. 1999.

[25] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proc. of the 27th Annual Intnl. Symp. on Computer Architecture*, pages 83–94, June 2000.

[26] D. C. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-1997-1342, Computer Sciences Department, University of Wisconsin–Madison, 1997.

[27] A. Buyuktosunoglu, D. Albonesi, S. Schuster, D. Brooks, P. Bose, and P. Cook. A circuit level implementation of an adaptive issue queue for power-aware microprocessors. In *Great Lakes Symposium on VLSI Design*, pages 73–78, 2001.

[28] H. Cain and P. Nagpurkar. Runahead execution vs. conventional data prefetching in the IBM POWER6 microprocessor. In *2010 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2010.

[29] H. W. Cain and M. H. Lipasti. Memory Ordering: A Value-Based Approach. In *Proc. of the 31st Annual Intnl. Symp. on Computer Architecture*, June 2004.

[30] R. Canal, J.-M. Parcerisa, and A. Gonzalez. A Cost-Effective Clustered Architecture. In *Proc. of the Intnl. Conf. on Parallel Architectures and Compilation Techniques*, Oct. 1999.

[31] K. Chakraborty. *Over-provisioned Multicore Systems*. PhD thesis, University of Wisconsin, 2008.

[32] K. Chakraborty, P. M. Wells, and G. S. Sohi. Computation Spreading: Employing Hardware Migration to Specialize CMP Cores On-the-fly. In *Proc. of the 12th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[33] J. Chang and G. S. Sohi. Cooperative Caching for Chip Multiprocessors. In *Proc. of the 33nd Annual Intnl. Symp. on Computer Architecture*, June 2006.

[34] J. Charles, P. Jassi, A. N. S, A. Sadat, and A. Fedorova. Evaluation of the Intel Core i7 Turbo Boost feature. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Oct. 2009.

[35] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Simultaneous Speculative Threading: A Novel Pipeline Architecture Implemented in Sun's ROCK Processor. In *Proc. of the 36th Annual Intnl. Symp. on Computer Architecture*, June 2009.

[36] J.-D. Choi and H. Srinivasan. Deterministic Replay of Java Multithread Applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT-98)*, pages 48–59, Aug. 1998.

[37] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture Optimizations for Exploiting Memory-Level Parallelism. In *Proc. of the 31st Annual Intnl. Symp. on Computer Architecture*, pages 76–87, June 2004.

[38] G. Z. Chrysos and J. S. Emer. Memory Dependence Prediction using Store Sets. In *Proc. of the 26th Annual Intnl. Symp. on Computer Architecture*, pages 142–153, May 1999.

[39] A. Cristal, O. J. Santana, F. Cazolra, M. Galluzi, T. Ramirez, M. Percias, and M. Valero. Kilo-instruction processors: overcoming the memory wall. *IEEE Micro*, 25(3), May 2005.

[40] A. Cristal, O. J. Santana, M. Valero, and J. F. Martinez. Toward kilo-instruction processors. *ACM Transactions on Architecture and Compiler Optimizations*, 1(4), Dec. 2004.

[41] Z. Cvetanovic. Performance analysis of the Alpha 21364-based HP GS1280 multiprocessor. In *Proc. of the 30th Annual Intnl. Symp. on Computer Architecture*, pages 218–229, June 2003.

[42] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of the 6th USENIX Symp. on Operating Systems Design and Implementation*, Dec. 2004.

[43] B. S. Deepaksubramanyan and A. Nunez. Analysis of Subthreshold Leakage Reduction in CMOS Digital Circuits. In *13th NASA VLSI Symposium*, 2007.

[44] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *Proc. of the 29th Annual Intnl. Symp. on Computer Architecture*, May 2002.

[45] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, G. Semeraro, G. Magklis, and M. L. Scott. Integrating Adaptive On-Chip Storage Structures for Reduced Dynamic Power. In *Proc. of the Intnl. Conf. on Parallel Architectures and Compilation Techniques*, pages 141–152, Sept. 2002.

[46] J. Dundas and T. Mudge. Improving Data Cache Performance by Pre-Executing Instructions Under a Cache Miss. In *Proc. of the 1997 Intnl. Conf. on Supercomputing*, pages 68–75, July 1997.

[47] A. N. Eden and T. Mudge. The YAGS Branch Prediction Scheme. In *Proc. of the 25th Annual Intnl. Symp. on Computer Architecture*, pages 69–77, June 1998.

[48] W. T. F. Encyclopedia. IBM POWER. http://en.wikipedia.org/wiki/IBM_POWER.

[49] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The Multicluster Architecture: Reducing Processor Cycle Time Through Partitioning. *International Journal of Parallel Programming*, 27(5):327–356, Oct. 1999.

[50] D. Folegnani and A. Gonzalez. Energy-effective issue logic. In *Proc. of the 28th Annual Intnl. Symp. on Computer Architecture*, pages 230–239, July 2001.

[51] I. T. R. for Semiconductors. ITRS 2006 Update. Semiconductor Industry Association, 2006. http://www.itrs.net/Links/2006Update/2006UpdateFinal.htm.

[52] I. T. R. for Semiconductors. ITRS 2007 Update. Semiconductor Industry Association, 2007. http://www.itrs.net/Links/2007ITRS/Home2007.htm.

[53] M. Franklin. Multi-Version Caches for Multiscalar Processors. In *In Proceedings of the First International Conference on High Performance Computing (HiPC)*, 1995.

[54] M. Franklin and G. S. Sohi. The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism. In *Proc. of the 19th Annual Intnl. Symp. on Computer Architecture*, pages 58–67, May 1992.

[55] M. Franklin and G. S. Sohi. ARB: A Hardware Mechanism for Dynamic Reordering of Memory References. *IEEE Transactions on Computers*, 45(5):552–571, May 1996.

[56] A. Gandhi, H. Akkary, R. Rajwar, S. Srinivasan, and K. Lai. Scalable Load and Store Processing in Latency Tolerant Processors. In *Proc. of the 32nd Annual Intnl. Symp. on Computer Architecture*, June 2005.

[57] B. Ganesh, A. Jalell, D. Wang, and B. Jacob. Fully-buffered dimm memory architectures: Understanding mechanisms, overheads and scaling. In *Proc. of the 13th IEEE Symp. on High-Performance Computer Architecture*, Feb. 2007.

[58] G. Gerosa, S. Curtis, M. D'Addeo, B. Jiang, B. Kuttanna, F. Merchant, B. Patel, M. Taufique, and H. Samarchi. A Sub-2 W Low Power IA Processor for Mobile Internet Devices in 45 nm High-k Metal Gate CMOS. *IEEE Journal of Solid-State Circuits*, 44(1):73–82, 2009.

[59] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proc. of the 19th ACM Symp. on Operating System Principles*, Oct. 2003.

[60] D. Gibson and D. A. Wood. Forwardflow: A Scalable Core for Power-Constrained CMPs. In *Proc. of the 37th Annual Intnl. Symp. on Computer Architecture*, June 2010.

[61] C. Gniady, B. Falsafi, and T. Vijaykumar. Is SC + ILP = RC? In *Proc. of the 26th Annual Intnl. Symp. on Computer Architecture*, pages 162–171, May 1999.

[62] R. Gonzalez, A. Cristal, M. Percias, M. Valero, and A. Veidenbaum. An asymmetric clustered processor based on value content. In *Proc. of the 19th Intnl. Conf. on Supercomputing*, June 2005.

[63] J. Gonz lez and A. Gonz lez. Dynamic Cluster Resizing. In *Proceedings of the 21st International Conference on Computer Design*, 2003.

[64] P. Gratz, K. Sankaralingam, H. Hanson, P. Shivakumar, R. McDonald, S. Keckler, and D. Burger. Implementation and Evaluation of a Dynamically Routed Processor Operand Network. *Networks-on-Chip, 2007. NOCS 2007. First International Symposium on*, 2007.

[65] T. Halfhill. Transmeta Breaks x86 Low-Power Barrier. *Microprocessor Report*, pages 9–18, February 2000.

[66] L. Hammond, B. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with Transactional Coherence and Consistency (TCC). In *Proc. of the 11th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2004.

[67] A. Hartstein and T. R. Puzak. Optimum Power/Performance Pipeline Depth. In *Proc. of the 36th Annual IEEE/ACM International Symp. on Microarchitecture*, Dec. 2003.

[68] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, third edition, 2003.

[69] J. L. Henning. SPEC CPU2006 Benchmark Descriptions. *Computer Architecture News*, 34(4):1–17, 2006.

[70] A. Henstrom. US Patent #6,557,095: Scheduling operations using a dependency matrix, Dec. 1999.

[71] M. Herlihy. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, 2003.

[72] M. D. Hill. Multiprocessors Should Support Simple Memory Consistency Models. *IEEE Computer*, 31(8):28–34, Aug. 1998.

[73] M. D. Hill and M. R. Marty. Amdahl's Law in the Multicore Era. *IEEE Computer*, pages 33–38, July 2008.

[74] A. Hilton, S. Nagarakatte, and A. Roth. iCFP: Tolerating All-Level Cache Misses in In-Order Pipelines. In *Proc. of the 15th IEEE Symp. on High-Performance Computer Architecture*, Feb. 2009.

[75] T. Horel and G. Lauterbach. UltraSPARC-III: Designing Third Generation 64-Bit Performance. *IEEE Micro*, 19(3):73–85, May/June 1999.

[76] M. Horowitz, C.-K. K. Yang, and S. Sidiropoulos. High-Speed Electrical Signaling: Overview and Limitations. *IEEE Micro*, 18(1), Jan/Feb 1998.

[77] D. R. Hower and M. D. Hill. Rerun: Exploiting Episodes for Lightweight Race Recording. In *Proc. of the 35th Annual Intnl. Symp. on Computer Architecture*, June 2008.

[78] M. Huang, J. Renau, and J. Torrellas. Energy-efficient hybrid wakeup logic. In *ISLPED '02: Proceedings of the 2002 international symposium on Low power electronics and design*, pages 196–201, New York, NY, USA, 2002. ACM.

[79] M. Huang, J. Renau, S.-M. Yoo, and J. Torrellas. A framework for dynamic energy efficiency and temperature management. In *Proc. of the 33rd Annual IEEE/ACM International Symp. on Microarchitecture*, Dec. 2000.

[80] M. Huang, J. Renau, S. M. Yoo, and J. Torrellas. A framework for dynamic energy efficiency and temperature management. In *Proc. of the 33rd Annual IEEE/ACM International Symp. on Microarchitecture*, Dec. 2000.

[81] M. C. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: application to energy reduction. *Computer Architecture News*, 31(2), 2003.

[82] Intel, editor. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 3A: System Programming Guide Part 1. Intel Corporation.

[83] Intel. Microprocessor Quick Reference Guide. http://www.intel.com/pressroom/kits/quickref.htm.

[84] Intel. First the Tick, Now the Tock: Next Generation IntelÆ Microarchitecture (Nehalem). http://www.intel.com/technology/architecture-silicon/next-gen/whitepaper.pd% f, 2008.

[85] Intel. Intel and Core i7 (Nehalem) Dynamic Power Management, 2008.

[86] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez. Core Fusion: Accomodating Software Diversity in Chip Multiprocessors. In *Proc. of the 34th Annual Intnl. Symp. on Computer Architecture*, June 2007.

[87] A. Iyer and D. Marculescu. Power aware microarchitecture resource scaling. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, 2001.

[88] A. Jain and et al. A 1.2GHz Alpha Microprocessor with 44.8GB/s Chip Pin Bandwidth. In *Proceedings of the IEEE 2001 International Solid-State Circuits Conference*, pages 240–241, 2001.

[89] R. Joseph and M. Martonosi. Run-time power estimation in high performance microprocessors. In *ISLPED '01: Proceedings of the 2001 international symposium on Low power electronics and design*, pages 135–140, 2001.

[90] N. P. Jouppi and D. W. Wall. Available instruction-level parallelism for superscalar and super-pipelined machines. In *Proc. of the 3rd Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Apr. 1989.

[91] D. Kanter. The Common System Interface: Intel's Future Interconnect. http://www.realworldtech.com/page.cfm?ArticleID=RWT082807020032.

[92] S. Keckler, D. Burger, K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. Govindan, P. Gratz, D. Gulati, H. H. amd C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, and P. Shivakumar. *Architecture and Implementation of the TRIPS Processor*. CRC Press, 2007.

[93] R. M. Keller. Look-Ahead Processors. *ACM Computing Surveys*, 7(4), 1975.

[94] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler. Composable Lightweight Processors. In *Proc. of the 40th Annual IEEE/ACM International Symp. on Microarchitecture*, Dec. 2007.

[95] I. Kim and M. H. Lipasti. Half-price architecture. In *Proc. of the 30th Annual Intnl. Symp. on Computer Architecture*, pages 28–38, June 2003.

[96] L. Kleinrock. *Queueing Systems Volume I: Theory*. Wiley Interscience, 1975.

[97] G. Konstadinidis, M. Tremblay, S. Chaudhry, M. Rashid, P. Lai, Y. Otaguro, Y. Orginos, S. Parampalli, M. Steigerwald, S. Gundala, R. Pyapali, L. Rarick, I. Elkin, Y. Ge, and I. Parulkar. Architecture and Physical Implementation of a Third Generation 65 nm, 16 Core, 32 Thread Chip-Multithreading SPARC Processor. *IEEE Journal of Solid-State Circuits*, 44(1):7–17, 2009.

[98] G. K. Konstadinidis and et al. Implementation of a Third-Generation 1.1-GHz 64-bit Microprocessor. *IEEE Journal of Solid-State Circuits*, 37(11):1461–1469, Nov 2002.

[99] R. Kumar, D. M. Tullsen, and N. P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *Proc. of the Intnl. Conf. on Parallel Architectures and Compilation Techniques*, Sept. 2006.

[100] L. Lamport. How to Make a Multiprocess Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, pages 690–691, 1979.

[101] A. R. Lebeck, T. Li, E. Rotenberg, J. Koppanalil, and J. P. Patwardhan. A Large, Fast Instruction Window for Tolerating Cache Misses. In *Proc. of the 29th Annual Intnl. Symp. on Computer Architecture*, May 2002.

[102] B. Liblit. An Operational Semantics for LogTM. Technical Report 1571, Computer Sciences Department, University of Wisconsin–Madison, Aug. 2006.

[103] P. S. Magnusson et al. SimICS/sun4m: A Virtual Workstation. In *Proceedings of Usenix Annual Technical Conference*, June 1998.

[104] P. S. Magnusson et al. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[105] D. Markovic, B. Nikolic, and R. Brodersen. Analysis and design of low-energy flip-flops. In *ISLPED '01: Proceedings of the 2001 international symposium on Low power electronics and design*, 2001.

[106] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, pages 92–99, Sept. 2005.

[107] J. Martinez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors. In *Proc. of the 35th Annual IEEE/ACM International Symp. on Microarchitecture*, Nov. 2002.

[108] M. R. Marty and M. D. Hill. Virtual Hierarchies to Support Server Consolidation. In *Proc. of the 34th Annual Intnl. Symp. on Computer Architecture*, June 2007.

[109] S. Mathew, M. Anders, B. Bloechel, T. Nguyen, R. Krishnamurthy, and S. Borkar. A 4-GHz 300-mW 64-bit integer execution ALU with dual supply voltages in 90-nm CMOS. *IEEE Journal of Solid-State Circuits*, 40(1):44–51, 2005.

[110] A. McDonald, J. Chung, B. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural Semantics for Practical Transactional Memory. In *Proc. of the 33nd Annual Intnl. Symp. on Computer Architecture*, June 2006.

[111] R. McDougall and J. Mauro. *Solaris internals: Solaris 10 and OpenSolaris kernel architecture*. Sun Microsystems Pressslash Prentice Hall, Upper Saddle River, NJ, USA, second edition, 2007.

[112] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[113] J. F. M. Meyrem Kyrman, Nevin Kyrman. Cherry-MP: Correctly Integrating Checkpointed Early Resource Recycling in Chip Multiprocessors. In *Proc. of the 38th Annual IEEE/ACM International Symp. on Microarchitecture*, Nov. 2005.

[114] M. Moir. Practical Implementations of Non-blocking Synchronization Primitives. In *Sixteenth ACM Symposium on Principles of Distributed Computing, Santa Barbara, California*, Aug. 1997.

[115] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *Proc. of the 35th Annual Intnl. Symp. on Computer Architecture*, June 2008.

[116] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, pages 114–117, Apr. 1965.

[117] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-Based Transactional Memory. In *Proc. of the 12th IEEE Symp. on High-Performance Computer Architecture*, pages 258–269, Feb. 2006.

[118] A. Moshovos, S. E. Breach, T. Vijaykumar, and G. S. Sohi. Dynamic Speculation and Synchronization of Data Dependences. In *Proc. of the 24th Annual Intnl. Symp. on Computer Architecture*, pages 181–193, June 1997.

[119] A. Moshovos and G. S. Sohi. Streamlining Inter-Operation Communication via Data Dependence Prediction. In *Proc. of the 30th Annual IEEE/ACM International Symp. on Microarchitecture*, pages 235–245, Dec. 1997.

[120] O. Mutlu, H. Kim, and Y. N. Patt. Techniques for Efficient Processing in Runahead Execution Engines. In *Proc. of the 32nd Annual Intnl. Symp. on Computer Architecture*, June 2005.

[121] O. Mutlu, H. Kim, and Y. N. Patt. Efficient Runahead Execution: Power-Efficient Memory Latency Tolerance. *IEEE Micro*, 26(1), Jan/Feb 2006.

[122] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead Execution: An Effective Alternative to Large Instruction Windows. *IEEE Micro*, 23(6):20–25, Nov/Dec 2003.

[123] R. Nagarajan, K. Sankaralingam, D. Burger, and S. Keckler. A Design Space Evaluation of Grid Processor Architectures. In *Proc. of the 34th Annual IEEE/ACM International Symp. on Microarchitecture*, pages 40–51, Dec. 2001.

[124] U. Nawathe, M. Hassan, K. Yen, A. Kumar, A. Ramachandran, and D. Greenhill. Implementation of an 8-Core, 64-Thread, Power-Efficient SPARC Server on a Chip. *IEEE Journal of Solid-State Circuits*, 43(1):6–20, January 2008.

[125] H.-J. Oh, S. Mueller, C. Jacobi, K. Tran, S. Cottier, B. Michael, H. Nishikawa, Y. Totsuka, T. Namatame, N. Yano, T. Machida, and S. Dhong. A fully pipelined single-precision floating-point unit in the synergistic processor element of a CELL processor. *IEEE Journal of Solid-State Circuits*, 41(4):759–771, 2006.

[126] S. Palacharla and J. E. Smith. Complexity-Effective Superscalar Processors. In *Proc. of the 24th Annual Intnl. Symp. on Computer Architecture*, pages 206–218, June 1997.

[127] I. Park, C. Ooi, and T. Vijaykumar. Reducing Design Complexity of the Load/Store Queue. In *Proc. of the 36th Annual IEEE/ACM International Symp. on Microarchitecture*, Dec. 2003.

[128] M. Percias, A. Cristal, R. Gonzalez, D. A. Jimenez, and M. Valero. A Decoupled KILO-Instruction Processor. In *Proc. of the 12th IEEE Symp. on High-Performance Computer Architecture*, Feb. 2006.

[129] D. Ponomarev, G. Kucuk, and K. Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *Proc. of the 34th Annual IEEE/ACM International Symp. on Microarchitecture*, Dec. 2001.

[130] C. V. Praun and X. Zhuang. Bufferless Transactional Memory with Runahead Execution, Jan. 2009. Pub. No. WO/2009/009583, International Application No. PCT/US2008/069513, on behalf of IBM.

[131] M. Prvulovic. CORD: Cost-effective (and nearly overhead-free) Order Recording and Data race detection. In *Proc. of the 12th IEEE Symp. on High-Performance Computer Architecture*, Feb. 2006.

[132] S. E. Raasch, N. L. Binkert, and S. K. Reinhardt. A scalable instruction queue design using dependence chains. In *Proc. of the 29th Annual Intnl. Symp. on Computer Architecture*, pages 318–329, May 2002.

[133] J. M. Rabaey. *Digital Integrated Circuits*. Prentence Hall, 1996.

[134] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proc. of the 34th Annual IEEE/ACM International Symp. on Microarchitecture*, Dec. 2001.

[135] R. Rajwar and J. R. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In *Proc. of the 10th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[136] M. A. Ramirez, A. Cristal, A. V. Veidenbaum, L. Villa, and M. Valero. Direct Instruction Wakeup for Out-of-Order Processors. In *IWIA '04: Proceedings of the Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA'04)*, pages 2–9, Washington, DC, USA, 2004. IEEE Computer Society.

[137] T. Ramirez, A. Pajuelo, O. J. Santana, and M. Valero. Runahead Threads to improve SMT performance. In *Proc. of the 14th IEEE Symp. on High-Performance Computer Architecture*, Feb. 2008.

[138] D. Risley. A CPU History. http://www.pcmech.com/article/a-cpu-history/.

[139] A. Roth. Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization. In *Proc. of the 32nd Annual Intnl. Symp. on Computer Architecture*, June 2005.

[140] S. M. Sait and H. Youssef. *VLSI Physical Design Automation: Theory and Practice*. World Scientific, 1999.

[141] P. Salverda and C. Zilles. A Criticality Analysis of Clustering in Superscalar Processors. In *Proc. of the 38th Annual IEEE/ACM International Symp. on Microarchitecture*, pages 228–241, Nov. 2005.

[142] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. Moore. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 422–433, June 2003.

[143] S. R. Sarangi, W. Liu, J. Torrellas, and Y. Zhou. ReSlice: Selective Re-Execution of Long-Retired Misspeculated Instructions Using Forward Slicing. In *Proc. of the 38th Annual IEEE/ACM International Symp. on Microarchitecture*, Nov. 2005.

[144] P. Sassone, J. R. II, E. Brekelbaum, G. Loh, and B. Black. Matrix Scheduler Reloaded. In *Proc. of the 34th Annual Intnl. Symp. on Computer Architecture*, pages 335–346, June 2007.

[145] A. S. Sedra and K. C. Smith. *Microelectronic Circuits, 4th Edition*. Oxford University Press, 1998.

[146] T. Sha, M. M. K. Martin, and A. Roth. NoSQ: Store-Load Communication without a Store Queue. In *Proc. of the 39th Annual IEEE/ACM International Symp. on Microarchitecture*, pages 285–296, Dec. 2006.

[147] T. Sherwood and B. Calder. Time Varying Behavior of Programs. Technical report, UC San Diego Technical Report UCSD-CS99-630, Aug. 1999.

[148] T. Sherwood, E. Perelman, and B. Calder. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. In *Proc. of the Intnl. Conf. on Parallel Architectures and Compilation Techniques*, pages 3–14, Sept. 2001.

[149] T. Shyamkumar, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. CACTI 5.1. Technical Report HPL-2008-20, Hewlett Packard Labs, 2008.

[150] D. Sima. The Design Space of Regsiter Renaming Techniques. *IEEE Micro*, 20(5), 2000.

[151] M. D. Smith, M. Johnson, and M. A. Horowitz. Limits on multiple instruction issue. In *Proc. of the 3rd Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Apr. 1989.

[152] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar Processors. In *Proc. of the 22nd Annual Intnl. Symp. on Computer Architecture*, pages 414–425, June 1995.

[153] G. S. Sohi. Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers. *IEEE Transactions on Computers*, pages 349–359, Mar. 1990.

[154] G. S. Sohi and S. Vajapeyam. Instruction Issue Logic for High-Performance Interruptable Pipelined Processors. In *Proc. of the 14th Annual Intnl. Symp. on Computer Architecture*, pages 27–34, June 1987.

[155] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. Fast Checkpoint/Recovery to Support Kilo-Instruction Speculation and Hardware Fault Tolerance. Technical Report 1420, Computer Sciences Department, University of Wisconsin–Madison, Oct. 2000.

[156] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual Flow Pipelines. In *Proc. of the 11th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2004.

[157] V. Srinivasan, D. Brooks, M. Gschwind, P. Bose, V. Zyuban, P. Strenski, and P. Emma. Optimizing pipelines for power and performance. Nov. 2002.

[158] S. Stone, K. Woley, and M. Frank. Address-Indexed Memory Disambiguation and Store-to-Load Forwarding. In *Proc. of the 38th Annual IEEE/ACM International Symp. on Microarchitecture*, Nov. 2005.

[159] S. Subramaniam and G. Loh. Store Vectors for Scalable Memory Dependence Prediction and Scheduling. In *Proc. of the 12th IEEE Symp. on High-Performance Computer Architecture*, Feb. 2006.

[160] S. Subramaniam and G. H. Loh. Fire-and-Forget: Load/Store Scheduling with No Store Queue at All. In *Proc. of the 39th Annual IEEE/ACM International Symp. on Microarchitecture*, pages 273–284, Dec. 2006.

[161] I. Sun Microsystems. Solaris 10 Reference Manual Collection: man pages section 2: System Calls. http://docs.sun.com/app/docs/doc/816-5167.

[162] H. Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), March 2005.

[163] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. WaveScalar. In *Proc. of the 36th Annual IEEE/ACM International Symp. on Microarchitecture*, pages 228–241, Dec. 2003.

[164] S. Tam, S. Rusu, J. Chang, S. Vora, B. Cherkauer, and D. Ayers. A 65nm 95W Dual-Core Multi-Threaded Xeon Processor with L3 Cache. In *Proc. of the 2006 IEEE Asian Solid-State Circuits Conference*, Nov. 2006.

[165] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. CACTI 4.0. Technical Report HPL-2006-86, Hewlett Packard Labs, June 2006.

[166] G. S. Tjaden and M. J. Flynn. Detection and Parallel Execution of Independent Instructions. *IEEE Transactions on Computers*, (10), 1970.

[167] M. Tremblay and S. Chaudhry. A Third-Generation 65nm 16-Core 32-Thread Plus 32-Scout-Thread CMT SPARC Processor. In *ISSCC Conference Proceedings*, 2008.

[168] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proc. of the 23th Annual Intnl. Symp. on Computer Architecture*, pages 191–202, May 1996.

[169] E. Tune, D. Liang, D. M. Tullsen, and B. Calder. Dynamic Prediction of Critical Path Instructions. In *Proc. of the 7th IEEE Symp. on High-Performance Computer Architecture*, pages 185–196, Jan. 2001.

[170] Intel Turbo Boost Technology in Intel Core Microarchitecture (Nehalem) Based Processors. http://download.intel.com/design/processor/applnots/320354.pdf, November 2008.

[171] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. In *Proceedings of the London Mathematical Society, Series 2, 42*, pages 230–265.

[172] F. Vandeputte, L. Eeckhout, and K. D. Bosschere. Exploiting program phase behavior for energy reduction on multi-configuration processors. *Journal of Systems Architecture:the EUROMICRO Journal*, 53(8):489–500, 2007.

[173] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation Cores: Reducing the Energy of Mature Computations. In *Proc. of the 9th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[174] T. N. Vijaykumar and G. S. Sohi. Task Selection for a Multiscalar Processor. In *Proc. of the 31st Annual IEEE/ACM International Symp. on Microarchitecture*, pages 81–92, Nov. 1998.

[175] Virtutech Inc. Simics Full System Simulator. http://www.simics.com/.

[176] R. Vivekanandham, B. Amrutur, and R. Govindarajan. A scalable low power issue queue for large instruction window processors. In *Proc. of the 20th Intnl. Conf. on Supercomputing*, pages 167–176, June 2006.

[177] J. von Neumann. First Draft of a Report on the EDVAC. Technical report, Moore School of Electrical Engineering, Univ. of Pennsylvania, 1945.

[178] D. Wall. Limits of Instruction Level Parallelism. In *Proc. of the 4th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Apr. 1991.

[179] H.-S. Wang, X. Zhu, L.-S. Peh, and S. Malik. Orion: A Power-Performance Simulator for Interconnection Networks. In *Proc. of the 35th Annual IEEE/ACM International Symp. on Microarchitecture*, pages 294–305, Nov. 2002.

[180] Y. Watanabe, J. D. Davis, and D. A. Wood. WiDGET: Wisconsin Decoupled Grid Execution Tiles. In *Proc. of the 37th Annual Intnl. Symp. on Computer Architecture*, June 2010.

[181] D. L. Weaver and T. Germond, editors. *SPARC Architecture Manual (Version 9)*. PTR Prentice Hall, 1994.

[182] P. M. Wells, K. Chakraborty, and G. S. Sohi. Hardware Support for Spin Management in Overcommitted Virtual Machines. In *Proc. of the Intnl. Conf. on Parallel Architectures and Compilation Techniques*, Sept. 2006.

[183] P. M. Wells and G. S. Sohi. Serializing Instructions in System-Intensive Workloads: Amdahl's Law Strikes Again. In *Proc. of the 14th IEEE Symp. on High-Performance Computer Architecture*, Feb. 2008.

[184] S. J. Wilton and N. P. Jouppi. An enhanced access and cycle time model for on-chip caches. *Tech report 93/5, DEC Western Research Lab*, 1994.

[185] Wisconsin Multifacet GEMS Simulator. http://www.cs.wisc.edu/gems/.

[186] Y. Wu and T. fook Ngai. Run-ahead program execution with value prediction, 2007. U.S. Patent 7,188,234.

[187] P. Xekalakis, N. Ioannou, and M. Cintra. Combining thread level speculation helper threads and runahead execution. In *Proc. of the 23rd Intnl. Conf. on Supercomputing*, June 2009.

[188] M. Xu, R. Bodik, and M. D. Hill. A "Flight Data Recorder" for Enabling Full-system Multiprocessor Deterministic Replay. In *Proc. of the 30th Annual Intnl. Symp. on Computer Architecture*, pages 122–133, June 2003.

[189] M. Xu, R. Bodik, and M. D. Hill. A Regulated Transitive Reduction (RTR) for Longer Memory Race Recording. In *Proc. of the 12th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 49–60, Oct. 2006.

[190] K. C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, Apr. 1996.

[191] B. Zhai, D. Blaauw, D. Sylvester, and K. Flaunter. Theoretical and Practical Limits of Dynamic Voltage Scaling. In *Proc. of the 41st Annual Design Automation Conference*, pages 868–873, June 2004.

[192] R. Zlatanovici, S. Kao, and B. Nikolic. Energy,ÄìDelay Optimization of 64-Bit Carry-Lookahead Adders With a 240 ps 90 nm CMOS Design Example. *IEEE Journal of Solid-State Circuits*, 44(2):569–583, February 2009.

Appendix A

Supplements for Chapter 3

This appendix contains the supplemental information for Chapter 3: Evaluation Methodology.

A.1 Inlined SPARCv9 Exceptions

Two different areas of the SPARCv9 exception set are modeled with hardware acceleration in this thesis: fill operations on the data translation lookaside buffer (D-TLB) and register window spill/fill exceptions. By providing hardware acceleration, the target machines are not strictly compliant to the SPARCv9 architectural specification [181]. However, it is assumed that these mechanisms can be deactivated if strict adherence is required. Moreover, it is not the intention of this work to strictly adhere to any given instruction set architecture (ISA), but rather to provide insight into the general behavior of processor cores. To this end, the features specific to SPARCv9 not present in a more commonplace ISA like x86 have been accelerated with hardware support. This not only generalizes the results of this thesis, but also provides substantial speedups in some cases. The details of the implementation of these hardware accelerations are presented below, and a brief analysis of the performance implications follows.

Overall, results from this work reflect the findings of Wells and Sohi [183].

A.2 Hardware-Assisted D-TLB Fill

The SPARCv9 specification requires that loads and stores which do not successfully translate their logical addresses to physical addresses do not appear to execute. Instead, the program counter, *PC* (and the SPARC-specific next program counter register, *NPC*), is set to execute a software D-TLB fill routine, in privileged mode [181]. By requiring a control transfer to privileged mode only on faulting memory accesses, a core implementing SPARCv9 discovers at execute-time whether or not implicit control flow predictions are correct (i.e., implicit speculation that memory operations will not fault). However, for the misprediction recovery approaches used in this work (fine-grained checkpointing), it is cost-prohibitive to checkpoint processor state on *every* implicit control prediction (i.e., any instruction capable of generating an exception—including all loads and stores). Instead, checkpoints are used for common-case mispredictions (e.g., explicitly predicted branches) [155, 3, 190], and other techniques (i.e., a complete pipeline flush) are used to resolve rarer cases, such as exceptions and external interrupts.

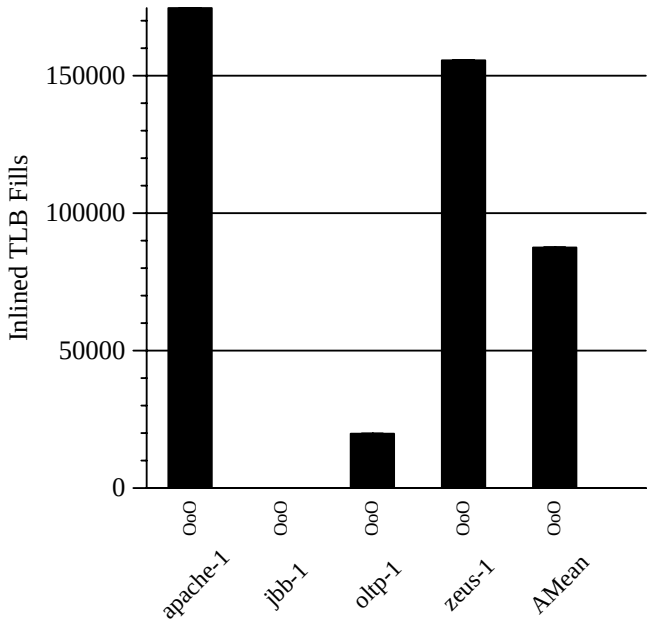


FIGURE A- 1 TLB misses in Commercial Workloads.

Unfortunately, a software-filled TLB pushes the boundaries of the (subjective) term *rare*. That is, some workloads exhibit sufficiently poor TLB locality such that TLB fills are, in fact, quite frequent. The most dramatic observed example is the `art` benchmark from SPEC OMP [16], which incurs roughly one TLB miss per 46 user-level instructions, though the Wisconsin Commercial Workload Suite [5] also exhibits high TLB miss rates, plotted in Figure A-1. The target machine uses a 1024-entry two-way set-associative D-TLB, augmented with a 16-entry fully-associative victim TLB. The cost of squashing a large pipeline to service these TLB fills can be noticeable, as not only must a processor enter and leave privileged mode (at least as expensive as a branch misprediction), but there is substantial opportunity cost in the form of independent instructions that may have been eligible for execution, had a full squash not been necessary.

In the Solaris operating system, the TLB fill routine attempts to service the miss by consulting a larger, software-managed translation table, the Translation Storage Buffer, or TSB [111, 161]. The common case of D-TLB fill is a fill that is entirely serviced via a lookup in the TSB.

Hardware-assisted D-TLB fill is modeled as a *hardware* D-TLB fill from the TSB. Because the TSB contains bits to identify the (rare) cases of unmapped or invalid accesses, accessing the TSB in hardware is sufficient to identify TLB fills that cannot be properly services by the TSB alone. It is

assumed that system software provides the TSB's location in physical memory to the acceleration hardware, and furthermore that this hardware can be disabled if a different TSB format is in use (or if a different translation structure is used). In other words, the hardware-assisted TLB fill evaluated in this design is specific to SPARCv9 running Solaris 9 and 10. Misses that cannot be serviced by the TSB trap to software as normal D-TLB misses.

The D-TLB fill hardware is located in the commit sequencing pipeline, which enables all microarchitectural models to leverage the same logic. Memory operations that fail translation (at execute-time) are artificially delayed as though simply incurring long delays in the memory hierarchy (e.g., with an MSHR). However, because these operations have already executed, they are not scheduler-resident. These operations wait until dedicated fill hardware detects that a stalled memory operation has reached the commit point (in a canonical out-of-order design, the head of the re-order buffer). At that time, the TSB is consulted to determine the correct translation, and the load (or store) is replayed via the commit-time replay circuitry, already in-place to facilitate speculative memory bypassing [146]. In the case of loads, the destination register is written, and the scheduler is notified that the load has completed execution. In the error-free case, the processor pipeline need not be squashed to service a D-TLB fill, allowing independent operations to execute opportunistically and out of order. Memory ordering is guaranteed through load and translation replay for vulnerable accesses [146].

A.3 Hardware-Assisted Inlined Register Spill, Fill, and Clean

SPARCv9 specifies 34 exceptions designed to deliver the appearance of an unbounded number of register windows to a user-space application, as well as to some portions of privileged code [181]. Of these exceptions, one exception type (*Flush*) is used to synchronize hardware register

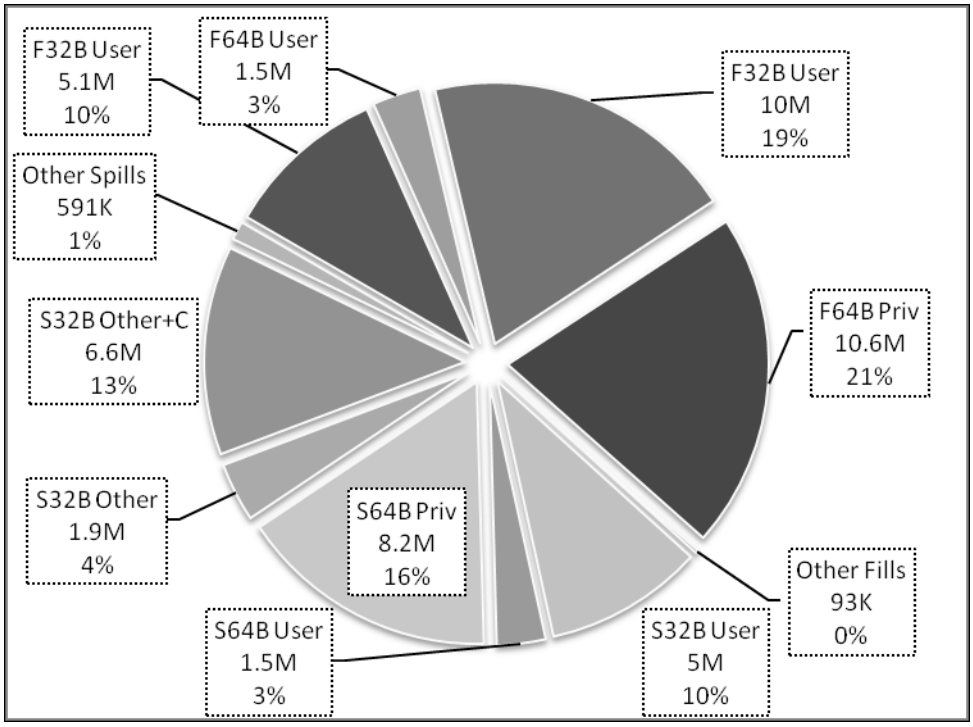


FIGURE A- 2 Register window exceptions by type, over all simulations.

state with reserved memory, and is relatively rare. A second exception type is used to guarantee isolation between threads (*Clean*), and is commonly coupled with other exceptions as an optimization (notably with spills). The remaining 32 exceptions are used by software to spill and fill registers. The exact exception type used depends on the contents of the SPARCv9 registers *WSTATE* and *OTHERWIN*, and is partially under software control (several different implementations of spill and fill traps are necessary, for instance, to support legacy SPARCv8 code, to support privileged windows spilled and filled by user processes, vice versa, to support mixed-mode (e.g., 32-bit and 64-bit) operation, and to enable certain optimizations, such as the combination of a clean window operation with a spill). Because software is free to implement these exception handlers in a flexible way, the generality of a single spill/fill acceleration mechanism is limited. Figure A-2 plots

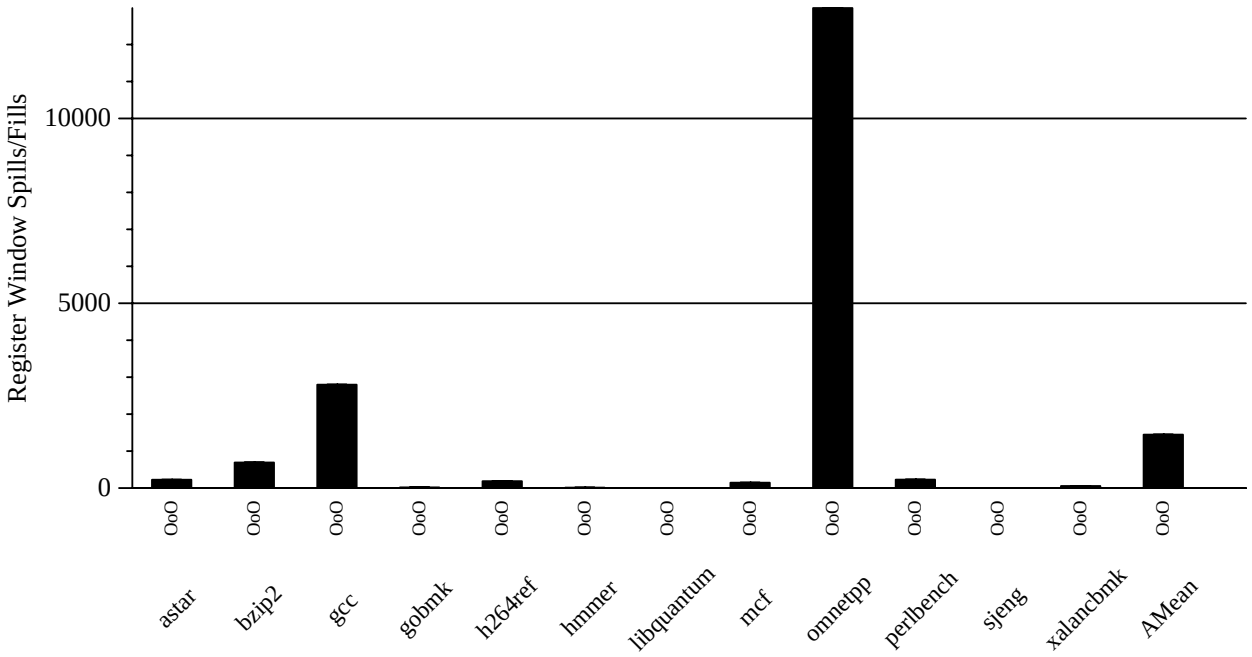


FIGURE A- 3 Spills and Fills in SPEC INT 2006.

the count of each spill and fill type encountered across the simulations leading to this thesis (counting individual benchmarks only once). As with TLB traps, the frequency of windowing traps varies by workload, with relatively few traps for most of the SPEC suites (e.g., Figure A-3 plots the combined number of spills and fills in 100-million instruction runs of SPEC INT 2006), and relatively frequent traps for commercial workloads (Figure A-4).

It is possible to detect likely spill/fill traps at decode-time, by recording the (speculative) state of windowing registers. Register spills and fills are the explicit result of some `save` and `restore` instructions, respectively, and the implicit result of some other opcodes that are similarly easy to detect at decode-time. To prevent these spills and fills, the acceleration hardware speculatively inserts operations fulfilling the requirements of a spill or fill handler into the predicted instruction stream at the appropriate position. It is assumed that the OS provides microcode implementing these operations at boot-time. The microcode for each trap type then resides in buffers dedicated

to that purpose. Collectively, these speculatively-inserted exception handlers are referred to as *inlined* spills and fills. Because these exceptions occur on *predicted* execution paths, inlined exceptions of this type are verified at commit-time. Inlined spills and fills are "best-effort"—they revert to explicit software control of exception handling when unhandled conditions arise (e.g., an unanticipated exception or external interrupt).

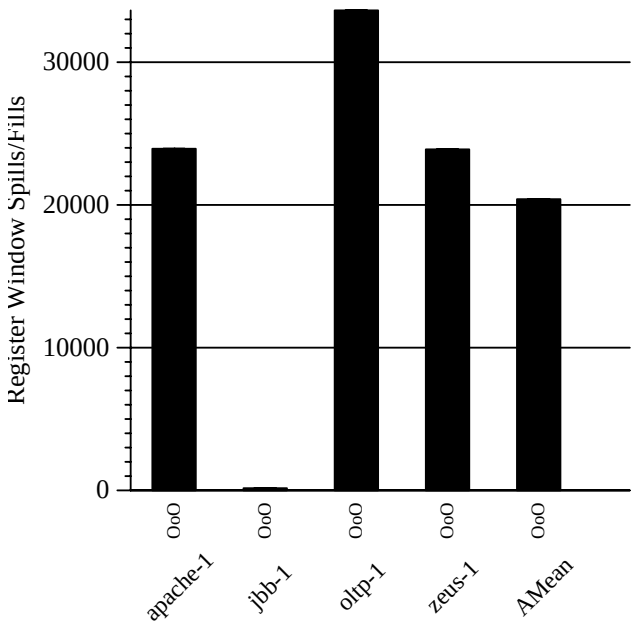


FIGURE A- 4 Spills and Fills in 100M instruction runs of the Wisconsin Commercial Workload suite.

Because this work does not explicitly simulate the boot process, nor model the necessary additions to the target's operating system to fill microcode tables with inlined exceptions, the first fill and spill exception of each type encountered during warmup phases of simulation is used as a template for all future invocations of the handler. In this manner, the hardware to inline exceptions dynamically bootstraps from observed execution stream. Subsequent executions of exceptions that diverge from this observed execution trigger software intervention. The bootstrapping process is repeated if several exceptions of the same type require software intervention with no intervening successful inlined exceptions (i.e., if the initial bootstrapping records a corner case of the exception handling code).

A.4 Results

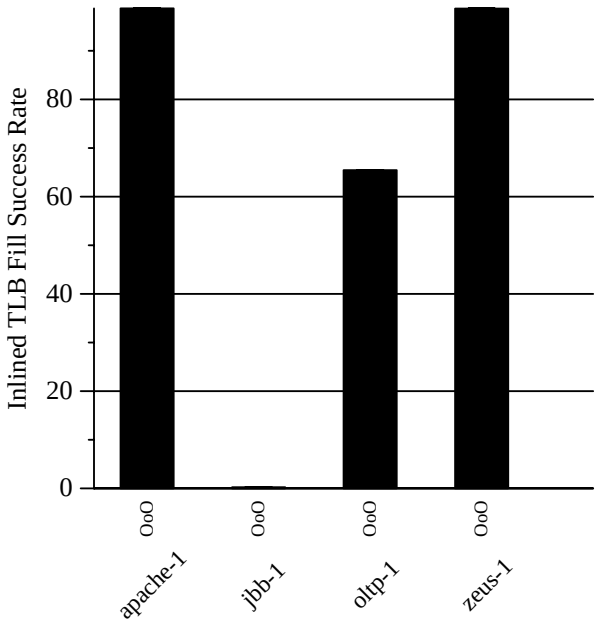


FIGURE A- 5 Success rate of TLB-fill inlining mechanism, commercial workloads.

Figure A-5 plots the success rate of the TLB fill inlining hardware for the commercial workload suite (the other suites are not as sensitive to TLB fill), as a percentage of all TLB exceptions. A *success* is considered an inlined exception that does not require a trap to software control. The most common condition under which TLB-fill inlining fails (e.g., for `jbb-1`), is a TSB miss. Failed exceptions trap to a software handler, as though no acceleration had been attempted. The inlining

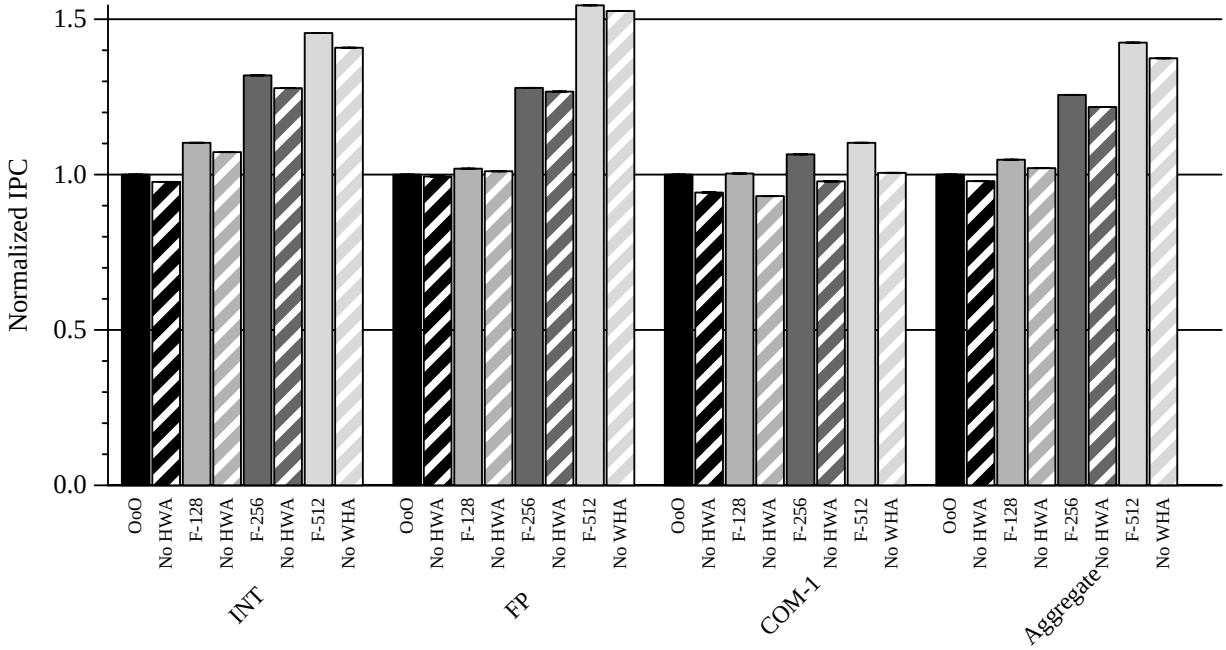


FIGURE A- 6 Normalized IPC across machine configurations, SPEC INT 2006 (INT), SPEC FP 2006 (FP), and Wisconsin Commercial Workloads (COM-1).

success rate of register window exceptions selected for acceleration is nearly 100%. Occasionally, software is invoked for exceptional cases, e.g., rare spill/fill types and nested exceptions.

Figure A-6 plots the resulting performance difference for a variety of machine configurations (refer to Chapter 3 for descriptions of these configurations—briefly, they include an out-of-order machine and Forwardflow configurations with varied instruction window sizes, each with and without hardware trap acceleration). I report **IPC** instead of runtime in the figure, due to a limitation of the methodology: The simulation infrastructure runs for a fixed number of committed instructions (or *transactions*, in some cases). However, speculatively inlined instructions do not count toward this execution goal. Therefore, hardware-accelerated configurations perform more total work than non-accelerated counterparts (using instruction commits as a measure of "work"), and take different execution paths. As such, runtimes are not directly comparable.

Appendix B

Supplements for Chapter 6

This appendix contains the supplemental information for Chapter 6: Scalable Cores in CMPs.

B.1 Power-Aware Microarchitecture

Section 6.3 discusses a procedure to discover energy consumption over a fixed interval using hardware event counters. Briefly, the linear sum of weighted event counts can be used to approximate. These weights are coefficients in a linear equation. In other words:

$$E_{execution} \approx E_{est} = \sum_{j \in C} N_j \cdot E_j^{est} \tag{B.1}$$

where E_j^{est} is the coefficient for counter j, and N_j is the event count.

I now present the method by which a hardware-based dynamic scaling policy, other hardware entity, or software via a procedure call, can leverage this mechanism to estimate energy consumption over an interval.

B.1.2 Platform Assumptions and Requirements

The method I suggest uses the underlying core's functional pathways and decode logic to directly compute the linear sum, above (Eqn. B.1). To do so in hardware, an instruction sequence is pre-loaded by the operating system, specialized for this purpose. I assume additional hardware

in the frontend logic, designed to insert these instructions into the instruction stream when signalled to do so by the hardware scaling policy (or other interested entity). Similar hardware already exists in the baselines used in this thesis, in the form of inlined exception handlers, and instruction decomposition logic for CISC operations. In particular, instructions are inserted predecode, as though they had been fetched, in program order, from the instruction cache. Alternatively, one could use a privileged software subroutine. In either case, instruction sequences interposed in the midst of an execution should be carefully designed to be completely transparent to a running program.

The inputs to this process are event counts, stored in special-purpose registers (i.e., performance counters), and coefficients, which I assume are stored in memory, but could also reside in special-purpose registers. I further assume these special-purpose registers are implemented by the decode logic, and are not precise with respect to the current commit point of the processor. However, I expect that they can be read without serializing the pipeline (in other words, software accounts for any inaccuracies in these counts due to reading them in a non-serialized fashion).

B.1.3 Power-Aware Architecture and Instruction Sequence

Before the energy measurement algorithm below can run successfully, system software must first make known the location of a per-configuration *Configuration Power Estimate Block* (CPEB), which records E_j^{est} for the currently-selected configuration. Each CPEB is 128 bytes in size (two cache lines), and consists of eight double-precision coefficients, as well as space to temporarily buffer the values of floating point registers (used for context save and restore, to ensure invisibility to software). The coefficients have been predetermined, at chip-design time, by the chip manufacturer.

Each processing core using this algorithm further requires an 8-byte memory area for storing the last observed timestamp (*LST*, i.e., a previous value of the `%tick` register). This area is not virtualized, but it must remain core-private for the duration of any measured execution. It, too, could be implemented as a special purpose register, though write precision is required. The addresses of the LST and CPEB must be physical: the algorithm below cannot tolerate a D-TLB miss on these accesses.

I assume the firmware has access to microarchitecture-specific hardware resources, such as processor-model-specific event counters, and microarchitecture specific addressing modes. In particular, the instruction sequence listed below may baffle a reader accustomed to SPARCv9-compliant assembly operations, e.g., by specifying local (integer) registers as source operands of floating-point conversion operations. However, these addressing modes are in line with the implementation of the microarchitectures used in this study, and are suitable for direct insertion into the second stage of the decode pipeline in all processor models used in this thesis.

Lastly, this implementation assumes a stack has been established, as is the common practice in SPARCv9, using the stack pointer register (`%sp`), and that this stack may be utilized by the power estimation microcode. This assumption is necessary because the inserted microcode uses register windows (`save` and `restore` opcodes) to preserve program state. The stack requirement can be relaxed, at the cost of expanding the CPEF by an additional 40 bytes, in order to explicitly save and restore local integer registers used by this computation. Power estimation is cancelled if the access to the stack pointer misses in the D-TLB.

Algorithm Description. The operation begins (line 03) by saving integer state, so as not to corrupt the measured execution. Lines 06 through 12 compute addresses of the CPEB and LST,

and issue prefetches to each cache block, in case these lines are not cached. Lines 13 through 15 perform a context save operation on floating-point state used by the algorithm. Context-restore occurs in lines 76-81.

In general, the computation performs three steps. First, coefficients are read from the CPEB, using an `ldqf` instruction to read two coefficients in parallel (e.g., Line 25 reads coefficients for the `store_instrs` and `fetch_puts` events). Second, event counts themselves are read from special-purpose registers (e.g., Line 27). Third event counts and coefficients are multiplied, and accumulated (in register `%F4`). To minimize the number of architectural registers saved and restored, these three steps are performed four times, in sequence (first in Lines 25-33, second in 35-44, etc.). This effectively decomposes the operation, based on coefficient position in the CPEB. Technically, the internal organization of the CPEB is implementation-dependent.

Static power consumption is addressed slightly differently. Line 60 reads the current time, and Lines 61 through 63 compute the cycles elapsed since the last measurement, and record the current time to the memory assigned to the LST. Thereafter, the number of elapsed cycles is used in a manner similar to that of the event counts themselves (i.e., `cycle_count` has a corresponding coefficient in the CPEB).

Once all contributions have been accumulated (in the sequence below, in `%F4`), a special-purpose instruction is used to communicate the consumed energy to the scaling policy (Line 72, a write to register `%energy_consumed`). I assume scaling policies will monitor this register for update. Other means for communicating the result are possible.

Instruction Sequence. Hereafter follows the actual instruction sequence used to measure energy for power-aware dynamic scaling policies. This constitutes the conclusion of this appendix.

```
01 # Context save:
02 # Need fresh locals and use of FP regs F0-F6 (double size)
03 save     %sp,   -208,       %sp
04
05 # Form address of constants
06 sethi    %hi(&CPEB)        %l7
07 addx     %l7,  %lo(&CPEB)  %l7     # l7 is address of CPEB
08 prefetch [%l7 + 0]                # Arrange to have next needed
09 prefetch [%l7 +64]                  # lines available
10 sethi    %hi(&LTS)         %l6
11 addx     %l6,   %lo(&LTS)  %l6     # l6 is address of LTS
12 prefetch [%l6 + 0] !Exclusive
13 stx      %fsr, [%l7 +88]           # FSR saved
14 stqf     %F0,  [%l7 +96]           # F0(Temp), F2(Temp) saved
15 stqf     %F4,  [%l7 +112]          # F4(Sum), F6(Temp) saved
16
17 # Implementing the linear sum
18 # General process:
19 # 1. Read coefficients (ldqf)
20 # 2. Read event counts (rd, partially covers ldqf latency)
21 # 3. Multiply and accumulate
22
23 # Process decomposed over pairs of coefficients,
24 # to optimize ldqfs
25 ldqf     [%l7 + 0],       %F0      # F0 = store_instrs_coeff
26                                    # F2 = fetch_puts_coeff
27 rd       %store_instrs,   %l0      # store_instrs
28 rd       %fetch_puts,     %l1      # fetch_puts
29 fitod    %l0,             %F6      # double(store_instrs)
30 fmuld    %F0,  %F6,       %F4      # F4 = 'Accumulated Energy'
31 fitod    %l0,             %F6      # double(fetch_puts)
32 fmuld    %F2,  %F6,       %F6      # double(correlated_energy)
33 faddd    %F4,  %F6,       %F4      # F4 = 'Accumulated Energy'
34
35 ldqf     [%l7 + 16],      %F0      # F0 = mshr_recycle_coeff
36                                    # F2 = l1i_access_coeff
37 rd       %mshr_recycles,  %l0      # mshr_recycles
38 rd       %l1i_access,     %l1      # l1i_access
39 fitod    %l0,             %F6      # double(mshr_recycles)
40 fmuld    %F0,  %F6,       %F6      # double(correlated_energy)
```

```
41 faddd    %F4,  %F6,       %F4      # F4 = 'Accumulated Energy'
42 fitod    %l2,             %F6      # double(l1i_access)
43 fmuld    %F2,  %F6,       %F6      # double(correlated_energy)
44 faddd    %F4,  %F6,       %F4      # F4 = 'Accumulated Energy'
45
46 ldqf     [%l7 + 32],      %F0      # F0 = l1d_access_coeff
47                                    # F2 = ialu_ops_coeff
48 rd       %l1d_accesss,    %l0      # l1d_accesss
49 rd       %ialu_ops,       %l1      # ialu_ops
50 fitod    %l0,             %F6      # double(l1d_accesss)
51 fmuld    %F0,  %F6,       %F6      # double(correlated_energy)
52 faddd    %F4,  %F6,       %F4      # F4 = 'Accumulated Energy'
53 fitod    %l2,             %F6      # double(ialu_ops)
54 fmuld    %F2,  %F6,       %F6      # double(correlated_energy)
55 faddd    %F4,  %F6,       %F4      # F4 = 'Accumulated Energy'
56
57 ldqf     [%l7 + 48],      %F0      # F0 = falu_ops_coeff
58                                    # F2 = cycle_count_coeff
59 rd       %falu_opss,      %l0      # falu_opss
60 rd       %tick,           %l1      # current time
61 ldx      [%l6 +  0],      %l2      # LST
62 stx      [%l6 +  0],      %l1      # Write current time to LST
63 subx     %l1,  %l2,       %l1      # cycle_count
64 fitod    %l0,             %F6      # double(falu_opss)
65 fmuld    %F0,  %F6,       %F6      # double(correlated_energy)
66 faddd    %F4,  %F6,       %F4      # F4 = 'Accumulated Energy'
67 fitod    %l2,             %F6      # double(cycle_count)
68 fmuld    %F2,  %F6,       %F6      # double(correlated_energy)
69 faddd    %F4,  %F6,       %F4      # F4 = 'Accumulated Energy'
70
71 # Pass result to power-aware compontents
72 wr       %F4,             %energy_consumed
73
74 # Context restore
75 # Restore FP regs F0-F6 (double size)
76 ldqf     [%l7 +96],       %F0      # F0, F2 restored
77 ldqf     [%l7 +112],      %F4      # F4, F6 restored
78 ldx      [%l7 +88],       %fsr     # FSR restored
79
80 # Restore previous window
81 restore  %sp, -208, %sp
```

Appendix C

Supplements for Chapter 5

This appendix presents the complete data from Chapter 5. Runtimes are listed in cycles. Categorized power consumption is normalized to that of *F-128*, as the absolute values produced by the WATTCH power models are nonsensical—only relative comparisons are valid between them.

TABLE B-1. Data tables.

FORWARDFLOW RUNTIMES

	F-32	F-64	F-128	F-256	F-512	F-1024
astar	159372601	120204074	98392526	92503185	90326357	89546028
bzip2	142935622	98906135	80937353	79041677	77167533	75869266
gcc	146277165	122338908	107009282	1028278	100838618	99427040
gobmk	126847363	10422075	88066570	84776863	83741764	83523828
h264ref	156305883	124224206	98027570	95763256	95221982	95091000
hmmer	93119436	71887576	56807506	51827561	49609554	46676647
libquantum	1232369835	651727125	384324608	199526264	110515309	68431995
mcf	92544223	662500779	539916301	484119275	454999941	444083808
omnetpp	130384069	104799448	84635178	82039774	80999432	80457401
perlbench	231558413	186175145	160399673	152922628	148524786	144319586
sjeng	121910265	99767606	81396736	78470738	78270511	78107556
xalancbmk	206279286	179237515	155573719	114548724	97188234	84558570
bwaves	625698987	400670272	236203198	174940001	122052831	69785262
cactus	139177765	112197116	70560713	64478432	56228308	50614168
calculix	90312168	82168391	49599602	43981071	42777893	42159466
dealII	117426205	98908267	80367800	74314517	71906502	67097030
gamess	86005720	68822675	48170099	45242258	44860773	44496037
gems	302434704	250094782	18786579	132390964	86089967	57928109
gromacs	138157229	130863068	110298092	99262502	94952759	91714606
lbm	659344757	460155376	322292704	225430666	157185303	117425804
leslie3d	157907765	134888297	119947356	116846157	115608184	115005882

milc	850611193	598360911	426173077	250753152	156970913	110555794
namd	111032308	80361087	60352116	56166260	56209143	55853530
povray	117225887	99037842	77496782	76052253	75985355	76050323
soplex	145558673	121066338	104211959	99886482	98969794	98448218
sphinx3	31864245	298987022	263611973	234716176	202723625	186841835
tonto	110495496	89621899	71278406	69136490	68625087	68824644
wrf	103606897	83244870	66918638	64672884	62028369	60911526
zeusmp	221031128	164936748	102293201	81874646	68488142	43808410
apache-1	362420242	305182476	267737916	253243985	244935985	243144788
jbb-1	163064813	137395919	119695034	11022192	105735781	10455294
oltp-1	185285988	161252686	142753177	137961538	135334892	134174349
zeus-1	36328931	308847456	27369436	255886483	245735143	242949041

BASELINE RUNTIMES

	OoO	CFP Runahead	
astar	108381104	103686496	10880601
bzip2	60309232	58827250	60706924
gcc	103529691	100804382	104823389
gobmk	89938113	87580945	91304136
h264ref	122608498	121435667	12303746
hmmer	47017620	41864209	47125729
libquantum	527382494	352733709	493723219
mcf	57006375	614510274	46781566
omnetpp	92934331	90016065	93143578
perlbench	165710376	154114025	167232988
sjeng	86053386	84874785	85649622
xalancbmk	157783493	128478845	127193514
bwaves	237979262	122006269	249311824
cactus	74774475	53855113	81181912
calculix	56461579	54831389	58326971
dealII	87835135	79762412	89185288
gamess	51866215	51054731	54796941
gems	19250011	10537161	21217322
gromacs	96610537	91313722	95930938
lbm	31359105	191502076	317022326
leslie3d	11917818	117339702	119582443
milc	450775123	287028237	477575518
namd	57276247	56770290	57348359

povray	70554865	72047663	71491300
soplex	105959499	10698802	10682845
sphinx3	277860842	242975507	256156718
tonto	73769757	74113551	76354584
wrf	67715834	64888739	67880278
zeusmp	109134815	60351753	113540517
apache-1	268842316	250167	160408691
jbb-1	118334495	108388549	123699305
oltp-1	144436017	138823309	146296835
zeus-1	275098282	25056042	197337592

NORMALIZED POWER BREAKDOWN FOR F-32

	Sched/UH	Bypass/ON	PRF/ARF	ALUs	L2/L3 DMEM	
astar	0.33	0.25	0.56	0.53	0.6	0.37
bzip2	0.44	0.51	0.56	0.69	0.77	0.52
gcc	0.5	0.6	0.74	0.66	0.75	0.53
gobmk	0.46	0.52	0.68	0.61	0.71	0.5
h264ref	0.62	1.33	0.61	0.55	0.72	0.49
hmmer	0.41	0.45	0.64	0.66	0.83	0.5
libquantum	0.3	0.4	0.31	0.34	0.33	0.25
mcf	0.39	0.44	0.57	0.53	0.59	0.42
omnetpp	0.48	0.58	0.65	0.6	0.67	0.47
perlbench	0.48	0.52	0.71	0.64	0.73	0.52
sjeng	0.42	0.42	0.65	0.6	0.69	0.46
xalancbmk	0.52	0.55	0.79	0.7	0.83	0.58
apache-1	0.46	0.57	0.69	0.54	0.68	0.49
jbb-1	0.47	0.48	0.75	0.66	0.77	0.55
oltp-1	0.53	0.61	0.78	0.65	0.77	0.55
zeus-1	0.46	0.59	0.7	0.54	0.68	0.5
	Rename/RCT	ROB/DQ	Other	Network	Fetch	Static/Clk
astar	0.38	0.37	0.37	0.8	0.49	0.98
bzip2	0.48	0.46	0.49	0.96	0.6	0.98
gcc	0.58	0.52	0.59	0.93	0.69	0.98
gobmk	0.51	0.48	0.51	0.97	0.66	0.98
h264ref	0.58	0.57	0.57	0.94	0.63	0.98
hmmer	0.57	0.53	0.58	0.83	0.64	0.98

libquantum	0.3	0.27	0.31	0.49	0.33	0.98
mcf	0.46	0.41	0.45	0.7	0.51	0.98
omnetpp	0.54	0.47	0.54	0.91	0.63	0.98
perlbench	0.56	0.5	0.57	0.82	0.68	0.98
sjeng	0.45	0.43	0.45	0.97	0.59	0.98
xalancbmk	0.64	0.54	0.63	0.82	0.73	0.98
apache-1	0.55	0.47	0.55	0.83	0.67	0.98
jbb-1	0.57	0.51	0.58	0.89	0.7	0.98
oltp-1	0.63	0.56	0.63	0.88	0.73	0.98
zeus-1	0.58	0.49	0.58	0.85	0.67	0.98

NORMALIZED POWER BREAKDOWN FOR F-64

	Sched/UH	Bypass/ON	PRF/ARF	ALUs	L2/L3 DMEM	
astar	0.64	0.63	0.77	0.62	0.79	0.53
bzip2	0.74	0.79	0.82	0.85	0.85	0.69
gcc	0.78	0.86	0.87	0.71	0.89	0.65
gobmk	0.77	0.88	0.84	0.68	0.88	0.63
h264ref	1.02	2.29	0.75	0.6	0.88	0.52
hmmer	0.72	0.78	0.81	0.73	0.91	0.62
libquantum	0.69	0.85	0.59	0.51	0.57	0.44
mcf	0.69	0.74	0.8	0.68	0.82	0.59
omnetpp	0.83	0.99	0.81	0.66	0.83	0.6
perlbench	0.8	0.89	0.87	0.71	0.89	0.65
sjeng	0.71	0.75	0.8	0.65	0.84	0.59
xalancbmk	0.79	0.9	0.88	0.71	0.92	0.66
apache-1	0.76	0.9	0.85	0.67	0.84	0.62
jbb-1	0.77	0.82	0.87	0.72	0.9	0.65
oltp-1	0.81	0.9	0.89	0.7	0.89	0.65
zeus-1	0.76	0.92	0.86	0.67	0.84	0.62

	Rename/RCT	ROB/DQ	Other	Network	Fetch	Static/Clk
astar	0.63	0.62	0.62	0.91	0.73	0.98
bzip2	0.74	0.71	0.74	0.98	0.79	0.98
gcc	0.8	0.74	0.8	0.97	0.87	0.98
gobmk	0.77	0.72	0.76	0.98	0.86	0.98
h264ref	0.98	0.83	0.97	0.95	0.88	0.98
hmmer	0.76	0.75	0.76	0.91	0.8	0.98
libquantum	0.58	0.55	0.58	0.69	0.6	0.98
mcf	0.72	0.67	0.7	0.87	0.76	0.98

omnetpp	0.78	0.72	0.78	0.95	0.82	0.98
perlbench	0.78	0.74	0.78	0.92	0.87	0.98
sjeng	0.69	0.67	0.69	0.98	0.8	0.98
xalancbmk	0.81	0.73	0.79	0.91	0.88	0.98
apache-1	0.78	0.71	0.78	0.92	0.85	0.98
jbb-1	0.77	0.73	0.77	0.94	0.86	0.98
oltp-1	0.83	0.76	0.82	0.94	0.88	0.98
zeus-1	0.8	0.71	0.79	0.93	0.85	0.98

NORMALIZED POWER BREAKDOWN FOR F-128

	Sched/UH	Bypass/ON	PRF/ARF	ALUs	L2/L3 DMEM	
astar	1	1	1	1	1	1
bzip2	1	1	1	1	1	1
gcc	1	1	1	1	1	1
gobmk	1	1	1	1	1	1
h264ref	1	1	1	1	1	1
hmmer	1	1	1	1	1	1
libquantum	1	1	1	1	1	1
mcf	1	1	1	1	1	1
omnetpp	1	1	1	1	1	1
perlbench	1	1	1	1	1	1
sjeng	1	1	1	1	1	1
xalancbmk	1	1	1	1	1	1
apache-1	1	1	1	1	1	1
jbb-1	1	1	1	1	1	1
oltp-1	1	1	1	1	1	1
zeus-1	1	1	1	1	1	1

	Rename/RCT	ROB/DQ	Other	Network	Fetch Static/Clk	
astar	1	1	1	1	1	1
bzip2	1	1	1	1	1	1
gcc	1	1	1	1	1	1
gobmk	1	1	1	1	1	1
h264ref	1	1	1	1	1	1
hmmer	1	1	1	1	1	1
libquantum	1	1	1	1	1	1
mcf	1	1	1	1	1	1
omnetpp	1	1	1	1	1	1
perlbench	1	1	1	1	1	1

sjeng	1	1	1	1	1	1
xalancbmk	1	1	1	1	1	1
apache-1	1	1	1	1	1	1
jbb-1	1	1	1	1	1	1
oltp-1	1	1	1	1	1	1
zeus-1	1	1	1	1	1	1

NORMALIZED POWER BREAKDOWN FOR F-256

	Sched/UH	Bypass/ON	PRF/ARF	ALUs	L2/L3 DMEM	
astar	1.22	1.13	1.09	1.17	1.09	1.12
bzip2	1.13	1.05	1.04	1.05	1.02	1.02
gcc	1.18	1.1	1.05	1.11	1.06	1.06
gobmk	1.15	1.06	1.04	1.12	1.04	1.05
h264ref	1.13	1.04	1.02	1.04	1.03	1.03
hmmer	1.23	1.07	1.05	1.19	1.06	1.09
libquantum	2.24	2.24	1.92	2.01	1.89	1.9
mcf	1.31	1.25	1.13	1.2	1.15	1.18
omnetpp	1.18	1.14	1.03	1.12	1.05	1.05
perlbench	1.23	1.17	1.05	1.18	1.1	1.1
sjeng	1.15	1.06	1.04	1.13	1.05	1.05
xalancbmk	1.64	1.56	1.36	1.49	1.39	1.43
apache-1	1.29	1.2	1.08	1.17	1.13	1.13
jbb-1	1.24	1.18	1.1	1.17	1.11	1.11
oltp-1	1.18	1.11	1.04	1.12	1.06	1.06
zeus-1	1.29	1.2	1.1	1.18	1.14	1.14

	Rename/RCT	ROB/DQ	Other	Network	Fetch	Static/Clk
astar	2.15	1.28	1.16	1.03	1.09	1.02
bzip2	2.13	1.23	1.14	1	1.09	1.02
gcc	2.06	1.29	1.1	1.01	1.07	1.02
gobmk	1.99	1.23	1.07	1.01	1.05	1.02
h264ref	1.91	1.1	1.03	1.01	1.03	1.02
hmmer	2.02	1.13	1.1	1.04	1.1	1.02
libquantum	3.59	2.35	1.92	1.69	1.94	1.02
mcf	2.3	1.42	1.23	1.08	1.2	1.02
omnetpp	2	1.29	1.07	1.01	1.05	1.02
perlbench	2.1	1.32	1.12	1.03	1.12	1.02
sjeng	2	1.23	1.07	1.01	1.05	1.02
xalancbmk	2.8	1.75	1.5	1.27	1.44	1.02

apache-1	2.17	1.38	1.17	1.04	1.13	1.02
jbb-1	2.24	1.37	1.2	1.04	1.14	1.02
oltp-1	2.04	1.28	1.09	1.03	1.06	1.02
zeus-1	2.17	1.38	1.17	1.05	1.14	1.02

NORMALIZED POWER BREAKDOWN FOR F-512

	Sched/UH	Bypass/ON	PRF/ARF	ALUs	L2/L3 DMEM	
astar	1.39	1.21	1.13	1.23	1.14	1.18
bzip2	1.27	1.11	1.1	1.09	1.06	1.06
gcc	1.37	1.21	1.08	1.17	1.11	1.12
gobmk	1.3	1.11	1.06	1.17	1.07	1.08
h264ref	1.25	1.07	1.03	1.05	1.04	1.04
hmmer	1.42	1.23	1.03	1.26	1.09	1.14
libquantum	4.8	4.87	3.47	3.67	3.35	3.38
mcf	1.66	1.51	1.22	1.38	1.25	1.34
omnetpp	1.32	1.19	1.05	1.15	1.08	1.08
perlbench	1.51	1.34	1.09	1.32	1.22	1.22
sjeng	1.3	1.12	1.05	1.17	1.1	1.1
xalancbmk	2.32	2.1	1.61	1.91	1.72	1.84
apache-1	1.61	1.42	1.15	1.34	1.25	1.24
jbb-1	1.46	1.3	1.16	1.26	1.2	1.2
oltp-1	1.36	1.19	1.07	1.18	1.1	1.11
zeus-1	1.63	1.43	1.17	1.35	1.27	1.26

	Rename/RCT	ROB/DQ	Other	Network	Fetch Static/Clk	
astar	2.01	1.63	1.22	1.05	1.14	1.06
bzip2	2.31	1.72	1.38	1.01	1.23	1.06
gcc	2.02	1.79	1.19	1.02	1.14	1.06
gobmk	1.87	1.6	1.12	1.01	1.08	1.06
h264ref	1.67	1.22	1.04	1.01	1.04	1.06
hmmer	1.8	1.2	1.15	1.06	1.15	1.06
libquantum	5.88	5.66	3.47	2.85	3.49	1.06
mcf	2.52	2.15	1.49	1.13	1.42	1.06
omnetpp	1.9	1.76	1.11	1.02	1.08	1.06
perlbench	2.19	1.92	1.28	1.05	1.26	1.06
sjeng	1.87	1.6	1.11	1.01	1.09	1.06
xalancbmk	3.55	3.01	2.06	1.45	1.85	1.06
apache-1	2.27	2.09	1.35	1.08	1.24	1.06
jbb-1	2.26	1.96	1.33	1.06	1.24	1.06

oltp-1	1.95	1.73	1.16	1.04	1.11	1.06
zeus-1	2.25	2.06	1.34	1.08	1.26	1.06

NORMALIZED POWER BREAKDOWN FOR F-1024

	Sched/UH	Bypass/ON	PRF/ARF	ALUs	L2/L3 DMEM	
astar	1.54	1.27	1.14	1.27	1.18	1.21
bzip2	1.44	1.2	1.17	1.13	1.1	1.1
gcc	1.56	1.29	1.11	1.22	1.16	1.17
gobmk	1.45	1.17	1.06	1.2	1.1	1.1
h264ref	1.36	1.1	1.03	1.05	1.05	1.04
hmmer	1.62	1.3	1.01	1.34	1.15	1.21
libquantum	8.58	8.2	5.59	6.05	4.8	5.08
mcf	1.92	1.65	1.27	1.48	1.29	1.42
omnetpp	1.44	1.22	1.06	1.17	1.09	1.09
perlbench	1.85	1.55	1.13	1.48	1.35	1.36
sjeng	1.48	1.2	1.05	1.24	1.16	1.16
xalancbmk	3.12	2.56	1.86	2.34	2.09	2.33
apache-1	1.86	1.54	1.17	1.45	1.31	1.29
jbb-1	1.62	1.36	1.18	1.3	1.25	1.23
oltp-1	1.54	1.26	1.08	1.23	1.13	1.14
zeus-1	1.89	1.58	1.19	1.48	1.35	1.33

	Rename/RCT	ROB/DQ	Other	Network	Fetch	Static/Clk
astar	2.15	2.3	1.27	1.05	1.19	1.14
bzip2	3.02	2.94	1.74	1.01	1.45	1.14
gcc	2.24	2.83	1.28	1.02	1.2	1.14
gobmk	1.98	2.35	1.16	1.01	1.11	1.14
h264ref	1.75	1.47	1.05	1.01	1.05	1.14
hmmer	1.99	1.32	1.22	1.1	1.22	1.14
libquantum	9.7	13.64	5.59	4.44	5.39	1.14
mcf	2.95	3.43	1.7	1.15	1.57	1.14
omnetpp	1.98	2.71	1.13	1.02	1.09	1.14
perlbench	2.6	3.26	1.48	1.07	1.4	1.14
sjeng	2.02	2.39	1.18	1.02	1.16	1.14
xalancbmk	5.02	5.82	2.83	1.63	2.37	1.14
apache-1	2.63	3.51	1.5	1.09	1.32	1.14
jbb-1	2.54	3.14	1.46	1.06	1.32	1.14
oltp-1	2.09	2.63	1.21	1.05	1.14	1.14
zeus-1	2.65	3.51	1.52	1.09	1.35	1.14

NORMALIZED POWER BREAKDOWN FOR CFP

	Sched/UH	Bypass/ON	PRF/ARF	ALUs	L2/L3 DMEM	
astar	2.11	144.45	7.66	1.28	1.1	1.04
bzip2	3.73	77.66	11.17	1.91	0.99	1.4
gcc	3.31	120.3	7.89	1.33	1.1	1.11
gobmk	2.19	174.33	8.14	1.38	1.05	1.05
h264ref	4.55	225.66	3.85	0.75	0.92	0.92
hmmer	3.38	178.85	8.89	1.76	1.32	1.35
libquantum	11.46	190.96	18.91	1.52	1.31	1.22
mcf	19.33	102.77	43.3	1.07	0.94	0.92
omnetpp	3.09	129.58	7.17	1.37	0.99	0.99
perlbench	6.36	142.88	12.7	1.58	1.27	1.27
sjeng	1.94	154.22	8.23	1.35	1.06	1.02
xalancbmk	5.02	177.1	13.83	1.55	1.3	1.29
apache-1	4.78	191.79	8.99	1.63	1.2	1.23
jbb-1	3.2	113.72	8.8	1.41	1.15	1.18
oltp-1	3.02	167.83	7.48	1.42	1.09	1.11
zeus-1	5.02	206.41	8.79	1.66	1.22	1.26
	Rename/RCT	ROB/DQ	Other	Network	Fetch	Static/Clk
astar	3.88	4.83	1.06	0.98	1.08	1.14
bzip2	5.35	5.52	1.38	1.04	1.26	1.14
gcc	4.74	5.37	1.1	1.02	1.13	1.14
gobmk	4.15	5.48	1.04	1	1.04	1.14
h264ref	2.66	4.53	1.09	0.96	0.94	1.14
hmmer	2.64	5.53	1.37	1.16	1.34	1.14
libquantum	6.25	16.34	1.1	1.07	1.49	1.14
mcf	8.15	24.74	0.9	0.91	0.93	1.14
omnetpp	4.97	4.59	1.01	0.99	0.97	1.14
perlbench	5.89	9.91	1.22	1.02	1.24	1.14
sjeng	3.87	4.92	0.98	1	1.02	1.14
xalancbmk	6.34	7.96	1.27	1.15	1.27	1.14
apache-1	5.72	7.65	1.28	1.06	1.2	1.14
jbb-1	4.78	5.58	1.12	1.04	1.13	1.14
oltp-1	4.61	5.8	1.1	1.02	1.08	1.14
zeus-1	5.86	7.47	1.31	1.06	1.22	1.14

NORMALIZED POWER BREAKDOWN FOR Runahead

	Sched/UH	Bypass/ON	PRF/ARF	ALUs	L2/L3 DMEM

astar	1.7	110.21	3.7	1.16	1	0.96
bzip2	3	60.09	5.41	1.79	0.94	1.32
gcc	2.55	94.49	3.75	1.26	1.04	1.05
gobmk	1.84	135.89	3.99	1.28	0.99	0.99
h264ref	4	185.66	2.01	0.74	0.9	0.91
hmmer	2.64	133.87	4.22	1.57	1.21	1.21
libquantum	4.59	151.72	4.14	2	1.18	1.63
mcf	2.77	138.75	6.04	1.88	1.71	1.89
omnetpp	2.57	103.99	3.57	1.31	0.95	0.95
perlbench	2.35	105.69	3.96	1.44	1.15	1.16
sjeng	1.68	120.97	4.1	1.27	0.98	0.95
xalancbmk	3.15	179.85	6.16	2.01	1.65	1.82
apache-1	3.74	161.94	4.6	1.62	1.14	1.4
jbb-1	2.22	83.88	3.79	1.25	1	1.03
oltp-1	2.37	129.32	3.53	1.32	1.01	1.04
zeus-1	2.92	149.51	3.16	1.47	1.03	1.07

	Rename/RCT	ROB/DQ	Other	Network	Fetch	Static/Clk
astar	2.85	2.31	0.96	0.95	0.98	1.14
bzip2	4.01	2.77	1.28	1.03	1.2	1.14
gcc	3.52	2.54	1.03	1	1.08	1.14
gobmk	3.12	2.56	0.96	0.99	0.98	1.14
h264ref	2.35	2.5	1.07	0.96	0.93	1.14
hmmer	2.31	2.8	1.21	1.09	1.21	1.14
libquantum	7.04	3.22	2.08	0.78	3.39	1.14
mcf	7.8	3.55	2.08	0.96	2.6	1.14
omnetpp	3.67	2.38	0.96	0.98	0.94	1.14
perlbench	3.89	2.8	1.11	0.97	1.18	1.14
sjeng	2.92	2.36	0.91	1	0.96	1.14
xalancbmk	6.82	3.53	1.81	1.05	2.03	1.14
apache-1	4.21	3.36	1.26	0.91	1.11	1.14
jbb-1	3.38	2.44	1	0.99	1.03	1.14
oltp-1	3.37	2.67	1.01	0.99	1.01	1.14
zeus-1	4.15	2.87	1.14	0.97	1.08	1.14

NORMALIZED POWER BREAKDOWN FOR OoO

	Sched/UH	Bypass/ON	PRF/ARF	ALUs	L2/L3	DMEM
astar	1.69	109.86	3.64	1.16	0.99	0.95
bzip2	2.97	59.39	5.28	1.77	0.93	1.31

gcc	2.53	93.57	3.67	1.25	1.03	1.04
gobmk	1.83	135.87	3.94	1.28	0.99	0.99
h264ref	4	185.26	1.97	0.74	0.9	0.9
hmmer	2.64	133.68	4.14	1.57	1.2	1.2
libquantum	2.52	82	2.33	1.02	0.76	0.76
mcf	1.57	82.91	3.83	1.11	1	0.97
omnetpp	2.55	103.2	3.49	1.3	0.95	0.94
perlbench	2.12	94.84	3.6	1.29	1.03	1.03
sjeng	1.65	119.05	3.99	1.25	0.96	0.94
xalancbmk	1.76	110.96	3.89	1.22	1.03	1.01
apache-1	2.55	124.98	3.3	1.28	1	1.04
jbb-1	2.24	83.76	3.77	1.26	1.01	1.04
oltp-1	2.31	126.97	3.42	1.29	1	1.02
zeus-1	2.55	130.38	3.08	1.27	0.99	1.03
	Rename/RCT	ROB/DQ	Other	Network	Fetch	Static/Clk
astar	1.99	2.31	0.95	0.95	0.98	1.14
bzip2	2.67	2.76	1.27	1.03	1.18	1.14
gcc	2.21	2.53	1.01	1.01	1.05	1.14
gobmk	2.06	2.57	0.96	1	0.98	1.14
h264ref	2.01	2.5	1.07	0.96	0.92	1.14
hmmer	2.22	2.79	1.21	1.09	1.2	1.14
libquantum	1.6	1.69	0.73	0.8	0.77	1.14
mcf	2	2.12	0.91	0.96	0.96	1.14
omnetpp	2.16	2.37	0.95	0.98	0.93	1.14
perlbench	2.11	2.62	0.97	0.98	1	1.14
sjeng	1.9	2.33	0.9	0.99	0.94	1.14
xalancbmk	2.15	2.24	0.97	0.99	0.99	1.14
apache-1	2.11	2.61	0.99	1	1	1.14
jbb-1	2.11	2.48	0.98	1	1	1.14
oltp-1	2.13	2.66	0.99	0.99	1	1.14
zeus-1	2.09	2.5	0.98	0.99	0.99	1.14