INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, Mi 48106-1346 USA
800-521-0600

An Analysis of Software Interface Issues for SMT Processors

Joshua Abram Redstone

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

Doctor of Philosophy

University of Washington

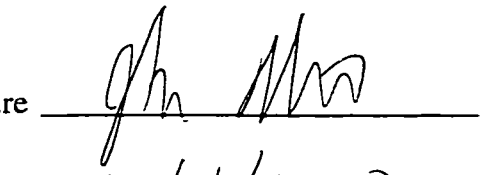
2002

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

UMI Number: 3072129

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of the dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to ProQuest Information and Learning, 300 North Zeeb Road, Ann Arbor, MI 48106-1346, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature 

Date 12/4/2002

University of Washington

Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Joshua Abram Redstone

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Co-Chairpersons of Supervisory Committee:

Henry Levy

Susan Eggers

Reading Committee:

Henry Levy

Susan Eggers

Jean-Loup Baer

Date: 12/4/2002

University of Washington

Abstract

An Analysis of Software Interface Issues for SMT Processors

by Joshua Abram Redstone

Co-Chairpersons of the Supervisory Committee:
Professor Henry Levy
Professor Susan Eggers
Department of Computer
Science and Engineering

Simultaneous Multithreading (SMT) has gradually progressed from a research concept to commercial processor technology. This thesis explores three software interface issues on SMT that are important to its real-world applicability. These issues are: operating system performance on SMT, the impact of spinning on SMT, and register file limitations to scaling SMT. We investigate these issues with a new, detailed simulation infrastructure capable of modeling all operating system activity.

First, we present an analysis of operating system execution on SMT. Many of the applications most amenable to multithreading technologies, such as the Apache web server, spend a significant fraction of their time in kernel code. We compare Apache's user- and kernel-mode behavior to a multiprogrammed SPECInt workload. Overall, our

results demonstrate the micro-architectural impact of an OS-intensive workload on an SMT processor. The synergy between the SMT processor and Web and OS software produces a greater throughput gain over superscalar execution than seen on any previously examined workloads, including commercial databases.

Second, we study the cost of synchronization on SMT. Spinning can exact a large performance cost on SMT, because all threads share execution resources. We quantify the impact of spinning on SMT and the performance benefit of replacing spinning with SMT-lock-based code. We observe that spinning's degradation of performance ranges widely between more than 3x on multiprogrammed workloads to a negligable amount on the Apache workload.

Finally, we explore architectural register sharing on SMT. A significant impediment to the construction of SMTs larger than two or four contexts is register file size. We introduce and evaluate mini-threads, a simple extension to SMT that increases thread-level parallelism without the commensurate increase in register hardware. A mini-threaded SMT CPU adds additional per-thread state to each hardware context; an application executing in a context can create mini-threads that will utilize its own per-thread state, but share the context's architectural register set. Our results quantify the factors affecting performance in detail and demonstrate that mini-threads can improve performance significantly, particularly on small-scale, space-sensitive CPU designs.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I am deeply grateful to all the people - colleagues, friends and family - that helped to make this dissertation possible. I would like to first thank my advisors, Hank Levy and Susan Eggers for believing in me, for generously and patiently guiding me through my graduate student tenure, and for sharing with me their rich and seasoned knowledge of how to conduct research. I'd also like to thank Jean-Loup Baer for serving on my reading committee.

This work has given me the privilege of working with many excellent people. Thanks to Larry Ruzzo for advising me on my qualifying project and also providing help with some of the statistical analysis in this work. Many thanks to the top-notch graduate students and faculty I have had the fortuity to learn from and enjoy friendship with, including Mike Swift, Brian Bershad, Marc Fiuczynski, Sujay Parekh, Matthai Philipose and E Chris Lewis.

Thanks also to the department staff for keeping me on track as a graduate student and for keeping the machines working. In particular, I would like to thank Frankye Jones and Lindsay Michimoto for their advice on negotiating the formalities of graduate school, Nancy Burr for helping us maintain the Alpha workstations, the support staff for patiently responding to my questions and problems, and Melody Kadenko-Ludwa for working her magic to manage the grants that helped to fund this research and quickly overcome any logistical or bureaucratic obstacle.

Finally, my deepest thanks to my friends and family that provided the non-academic sustenance to nourish my soul through graduate school. Thanks to my parents and brothers and sisters for their loving and tireless support. Thanks to my friends for helping me to lead a balanced and grounded life through graduate school.

This research was supported by NSF grants MIP-9632977 ITR-0085670, CCR-0121341, CCR00-85670, EIA96-32977 and DARPA grant F30602-97-2-0226. The simulations were partially executed on Alpha systems donated by Compaq. The SMT simulator

is based on an application-level simulator originally written by Dean Tullsen and modified by Sujay Parekh. Manu Thambi created the SMT web server support.

Chapter 1

Introduction

This thesis explores three important software interface issues on SMT. Simultaneous multithreading (SMT) [82] has emerged as an effective technique to dramatically increase processor throughput over a superscalar CPU. SMT maintains the state (e.g., registers and program counter) of multiple threads in hardware *contexts*, and fetches and executes instructions from multiple threads simultaneously each cycle. In effect, SMT converts the thread-level parallelism (TLP) of the individual threads scheduled on the processor into global, cross-thread instruction-level parallelism (ILP), leading to greater throughput. The threads executing on SMT share all pipeline resources, including the caches, introducing the potential for constructive and destructive interference.

SMT has gradually progressed from a research idea into a commercial processor technology. Despite this, three SMT software/hardware interface issues that are important to SMT performance remain unaddressed. The issues are: operating system performance on SMT, the impact of spinning on SMT, and architectural register limitations to scaling SMT. This thesis explores each of these issues in depth.

The first interface connects software to privileged hardware state via special instructions. Privileged state controls different aspects of thread execution, such as virtual address space protection (TLB), scheduling behavior (interrupts) and exception processing. It also provides mechanisms, such as flushing the caches or TLBs, that affect performance. The OS manages this interface on most machines, including SMT.

OS behavior can critically affect the performance of important SMT workloads both because it controls this interface and because it can consume a large fraction of processor resources. As a general-purpose throughput-enhancing mechanism, simultaneous multithreading is especially well suited to applications that are inherently multithreaded, such as database and Web servers. In server-based environments, the operating system can be a

crucial component of the workload; for example, our measurements show that the Apache Web server spends over 75% of its time in the kernel. However, OS performance on SMT lacks any empirical investigation.

Our investigation of OS performance on SMT seeks to answer several basic questions. First, how would previously reported application-only results change, if at all, when the operating system is added to the workload? In particular, we wish to verify whether the results of previous studies were overly optimistic by excluding the OS. Second, and more important, what are the key behavioral differences at the architectural level between an operating-system-intensive workload and a traditional (low-OS) workload, both executing on SMT? Third, how does an OS-intensive application like Apache benefit from SMT, and where does it spends its time from a software point of view? This analysis is interesting in its own right, because of the increasing importance of Web servers and similar applications. Overall, our results characterize both the architectural behavior and the software behavior (within the OS) of an OS-intensive workload, namely the Apache Web server.

The second software interface provides synchronization instructions that improve sychronization efficiency on SMT. Through these instructions contexts access SMT's hardware blocking locks (called SMT locks). SMT locks will block a context via hardware until a lock becomes available; the context consumes no execution resources during this time. The alternatives, the only options on most processors, are for threads to either spin while they wait for the lock or block in software and context switch to another thread.

SMT locks can potentially eliminate the need to spin on SMT. However, studying spinning is important for three reasons. First, spinning will continue to plague SMT workloads because of the impracticality of rewriting all code, including the kernel, to replace spinning with SMT-lock-based routines. Second, spinning can exact a large performance cost on SMT because all executing threads share pipeline resources and SMT's instruction-fetch chooser gives preference to high-throughput threads. Third, studying spinning provides insight into the more general question of the extent to which one thread's behavior affects other threads' performance on SMT.

We investigate the impact of spinning on SMT and quantify the performance benefit of replacing spinning with SMT-lock-based routines. We focus on spinning in one important piece of software, the operating system. Our study of OS performance, mentioned

above, specifies the set of minimal software changes required to convert a multiprocessor-aware OS so that it can execute on SMT. Our spinning study extends this to explore the issue of executing efficiently. We first observe that spinning only degrades performance to the extent that it occupies resources that non-spinning threads could have used effectively. The remainder of the study then develops and explores this observation in detail. The investigation consists of three parts. The first two explore performance characteristics of spinning threads on SMT, both in isolation and in combination with other non-spinning threads. The third part evaluates spinning's impact on many SMT/workload combinations by measuring the performance improvement of eliminating spinning.

The final software interface is the architectural register set that each SMT hardware context contains. Architectural registers provide software with a software-managed cache of values; most instruction set architectures consist of instructions optimized to operate on these values [69]. Architectures vary in the number of registers and applications vary in their register need.

We investigate architectural register organization and use, especially targeted towards small-scale SMTs. Recently, several manufacturers have announced two- to four-context SMTs, both as single CPUs and as components of multiple CPUs on a chip [39, 75]. While small-scale SMTs boost performance, they still leave modern wide-issue CPUs with underutilized resources, i.e., substantial performance potential is still unexploited. A primary obstacle to the construction of larger-scale SMTs is the register file. We propose and evaluate a new mechanism to boost thread-level parallelism (and therefore, throughput) on small-scale SMTs, without the commensurate increase in register file size. The mechanism, called *mini-threads*, alters the basic notion of a hardware context. On the hardware level, each context of a mini-thread enabled SMT processor contains additional thread state (e.g., an additional program counter) other than registers. The additional state allows an application to create more thread-level parallelism within a context by forking mini-threads that will *share* the context's architectural registers.

Mini-threads improve on traditional SMT processors in three ways. First, mini-threads conserve registers, because each mini-thread does not require a full architectural register set. Second, mini-thread allows *each* application the freedom to trade-off ILP for TLP within its hardware contexts. Third, in addition to the savings in registers, mini-

threads open up new possibilities for fine-grained thread programming. Each application can choose how to manage the architectural registers among the mini-threads that share it.

Mini-threads allow applications to make a trade-off between one thread per context that utilizes the full architectural register set, or multiple mini-threads per context, each accessing a subset of the architectural register set. Two factors determine the efficacy of this trade-off: (1) the performance *benefit* due to the extra mini-threads, and (2) the performance *degradation* due to fewer available registers per mini-thread. For each of these factors, there are also two levels of potential impact. At the hardware level, each can improve or degrade IPC, and at the software level, each can alter the number of instructions per unit of work. We evaluate each of these factors in detail; for example, we provide a detailed analysis of the changes in spill-code (which can increase as well as decrease!) due to reducing the number of registers.

This thesis addresses the above software interface issues with a detailed simulation infrastructure. We created an SMT simulator that executes the operating system by integrating a detailed, application-level SMT simulator into the SimOS machine simulation framework. The implementation presented unique challenges and involved a fair amount of software complexity. Our simulator infrastructure description includes a discussion of the most difficult challenges in integrating OS-support into an application-level simulator.

To summarize, the contributions of this thesis include the following:

- An infrastructure to model in detail both application and OS performance on SMT,
- An analysis of OS performance on SMT, focusing on Apache and more traditional user-level applications,
- An evaluation of spinning on SMT, and
- The proposal and evaluation of mini-threads on SMT.

The rest of this dissertation proceeds as follows. Chapter 2 introduces the simulation infrastructure common to all experiments in this thesis. This chapter discusses the challenges and lessons learned in building and using the infrastructure. Chapter 3 begins our

experimental results, covering operating system impact on SMT simulation. Chapter 4 examines spinning's impact on SMT and Chapter 5 evaluates mini-threads. We conclude and discuss future directions in Chapter 6. Related work discussions are included in the relevant chapters.

Chapter 2

Methodology

This thesis derives its results from simulations of SMT. This chapter introduces, in two parts, the simulation methodology common to all experiments: Section 2.1 describes the simulation infrastructure, and Section 2.2 discusses measurement methodology issues. The following chapters that cover our experimental results contain additional descriptions of methodology specific to those experiments.

2.1 Simulation infrastructure

Simulation lets us model future processors and capture detailed performance information about executing workloads. Current SMT simulation platforms capture application execution. However, the experiments in this thesis require modeling of the operating system in addition to user code. For this purpose no simulator existed.

At one level the OS is simply a large program. Its uniqueness lies in having access to low-level hardware resources (e.g., I/O device registers and internal CPU registers) and responding to low-level hardware events (e.g., exceptions and interrupts). Simulating the OS thus requires simulating those resources and events. For this research we integrated our SMT CPU simulator into the SimOS-Alpha hardware simulation framework [21]. This let us boot and run the operating system on the simulator and include in our simulation every instruction, privileged or non-privileged, that a real CPU would execute. The SimOS environment also executes Alpha PAL code, which is a layer of software below the operating system itself. PAL code is used, for example, to respond to TLB misses and handle synchronization within the OS, such as the SETIPL call.

This section covers different aspects of the simulator infrastructure. Section 2.1.1 describes the integration of the SMT processor simulator with SimOS and the additional wiring necessary to produce a complete simulator. Section 2.1.2 discusses issues in adapt-

ing a multiprocessor-aware OS to SMT. We list the base simulated processor parameters in Section 2.1.3, present the simulator workloads in Section 2.1.4, and describe additional infrastructure outside of the core processor model that facilitated our experiments in Section 2.1.5. Section 2.1.6 concludes with a discussion of some of the challenges and lessons learned from upgrading an application-level simulator to support the operating system.

2.1.1 Integrating SimOS and the SMT application-level simulator

The simulator we developed combines two existing pieces: an SMT application-level simulator and the SimOS machine simulator framework. Each provides a disjoint set of functions, so they require additional logic to function together. Figure 2.1 depicts the contribution of each to a complete machine model. We begin with a brief review of SMT (Section 2.1.1.1), then describe the SMT application-level simulator (Section 2.1.1.2) and the SimOS framework (Section 2.1.1.3), and finally cover the implementation "glue" that ties them together (Section 2.1.1.4).

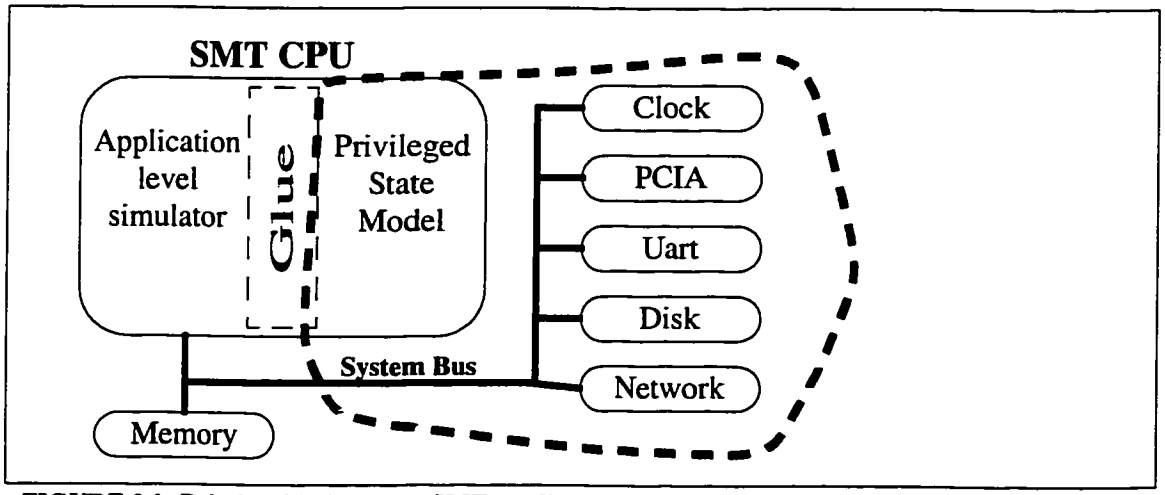


FIGURE 2.1: Relationship between SMT application-level simulator and SimOS. The figure depicts a complete machine. SimOS provides the components within the thick dashed line.

2.1.1.1 SMT background

SMT is a latency-tolerant CPU architecture that executes multiple instructions from multiple threads each cycle. The ability to issue instructions from different threads provides better utilization of execution resources by converting thread-level parallelism into

instruction-level parallelism. Previous research has established that SMT is effective in increasing throughput on a variety of workloads, while still providing good performance for single-threaded applications [81, 44, 45, 43, 87]. As a general-purpose throughput-enhancing mechanism, simultaneous multithreading is especially well suited to applications that are inherently multithreaded, such as database and Web servers, as well as multiprogrammed and parallel scientific workloads.

At the hardware level, SMT is a straightforward extension of modern, out-of-order superscalars, such as the MIPS R10000 [32] or the Alpha 21264 [33]. SMT duplicates the register file, program counter, subroutine stack and internal processor registers of a superscalar to hold the state of multiple threads (we call the set of hardware resources that contains the state of a thread a *context*). In addition to duplicating thread state, SMT has per-context mechanisms for pipeline flushing, instruction retirement, subroutine return prediction, and trapping. Intel estimates that the modifications to an out-of-order superscalar necessary to support a four-context SMT translated into only a 6% increase in chip area [27].

2.1.1.2 SMT simulator core

The SMT application-level simulator is a detailed, stand-alone, execution-based simulator used extensively in previous SMT studies [22, 43, 44, 45, 46, 47, 59, 77, 81, 82, 83]. It models the processor pipeline and memory system in great detail. While the simulator excels at modelling user-level code, it lacks the facilities necessary to accurately model an operating system.

On a real machine, implicit or explicit user requests for OS service begin with a trap, followed by a sequence of OS instructions that provide the service. Rather than model the trap and OS activity, the simulator crudely approximates both with a single, constant-cycle delay that is charged to the instruction causing the trap. Lacking physical addresses and a real TLB, the simulator models TLB misses in this fashion. Similarly, it supports a limited number of common system calls. For these, it charges a fixed cycle delay to the calling instruction and emulates the call's functionality. All other OS-invoking events -- such as TLB protection violations, all other system calls, synchronous traps and interrupts -- are not modeled and halt the simulator if they occur.

2.1.1.3 SimOS

SimOS [66] simulates all components of an SMP machine necessary to boot and run a commercial multiprocessor operating system. In this thesis we adapted SimOS-Alpha [21], a version of SimOS for the Alpha architecture that runs Compaq Tru64 Unix 4.0d. We discuss OS modifications for SMT in Section 2.1.2. Here, we summarize the hardware components modeled by SimOS. The non-microprocessor components that SimOS models include a clock, UARTS, DMA interfaces, network interfaces, disks, interrupt lines to the processor, and a bus (TLASER) that provides the processor with access to the devices.

SimOS provides a simple yet functionally complete processor simulator called 'Gamma.' The Gamma simulator models a very simple in-order pipeline that executes a single instruction each cycle. Functionally, this processor model differs from the SMT simulator solely because it models all privileged processor state. The Gamma simulator is based on the Alpha 21164 processor[1].

Modeling the privileged processor state requires support for the following:

- **Internal Processor Registers**: Seventy-four registers maintain the privileged processor state. Some of these registers, written by hardware, provide information on events such as TLB misses and faults, other protection violations, and interrupts. Other registers, written by the OS, provide control over the hardware, for example, by adding entries to or flushing the TLB or controlling interrupt and exception behavior.

- **Privileged instructions**: Five special instructions, accessible only from PAL code, define the interface through which software interacts with privileged hardware state. These instructions modify internal processor registers, allow unrestricted access to physical address space, and provide return-from-exception functionality.

- **Fully functional TLB**: The TLB maps virtual to physical addresses, checking protection on each access, and supports selective flushing of entries.

1. Privileged state varies slightly between processor incarnations.

- **Exceptions**: An exception includes any invalid operation, system request, or interrupt. Upon an exception, the simulator writes a description of the event into the appropriate internal processor registers and transfers control to a predetermined PAL entry PC based on the exception's cause. The Gamma simulator lacks a pipeline to flush before transferring control.

2.1.1.4 Additional logic required for integration

The previous section described missing SMT simulator pieces that are necessary to functionally model a processor in sufficient detail to execute an OS. Integrating the SMT simulator with SimOS involved incorporating these missing pieces into SMT's pipeline and memory system model. For example, the memory system simulator had to be augmented to support DMA transactions.

Supporting the non-processor components of SimOS required little modification to the SMT simulator: non-processor components interact with the processor mostly through special load/store instructions to the I/O address space. These instructions differ only slightly from normal load/store instructions, mostly in their issuance rules. Of the non-processor components, the DMA and network processor also interact with the memory system by reading/writing main memory. We added code to support these bus and memory transactions for DMA, but the network card transactions bypass the memory system simulator. Not modeling network transactions on the memory bus has no impact on results for all workloads except Apache because they do not use the network. Including network-related DMA transactions in the Apache workload would have had little impact on the memory bus, because memory bus delay averages only 0.25 cycles per bus transaction and adding network transactions would only double the total number of bus transactions.

The processor simulator required substantial modification to support the missing functions. We avoided reimplementing many of these functions by reusing code from the Gamma simulator. Most of our work focused on defining and implementing the correct pipeline behavior for each function. The following list addresses each component described in the previous section:

- **Internal Processor Registers**: Complex interactions occur between accesses to the IPRs and instruction flow through the pipeline (for example, memory operations cannot be reordered with respect to instructions that modify the dTLB). We based our instruction-ordering rules on the Alpha 21264, an out-of-order processor [19] (SMT is an out-of-order-processor).

- **Privileged instructions**: Privileged instructions have special requirements: they explicitly and implicitly affect many internal processor registers, and most cannot execute speculatively, because their effect cannot be undone. The IPRs do not support speculative modification. Therefore, the simulator must carefully manage when privileged instructions are issued relative to all other instructions that modify IPRs either explicitly (e.g., via an IPR store instruction) or implicitly (e.g., by trapping). We also model almost all OS/hardware interactions that affect the memory hierarchy, such as cache flush commands.

- **Fully functional TLB**: Supporting a real TLB increases the varieties of exceptions (described in the next paragraph). In addition, supporting physical addresses requires extensive modifications to the memory system simulator, which only approximated physical addresses by concatenating a thread-id to the virtual address. Full physical address simulation introduces aliasing problems absent from the application-level SMT simulator.

- **Exceptions:** Neither exception facilities nor a general pipeline flushing mechanism existed in the SMT stand-alone simulator. Supporting exceptions involved: (1) adding logic to detect all possible exceptions, (2) prioritizing exceptions consistently when many occur in a single cycle, (3) modeling the exception at the correct time (e.g., the correct pipeline stage), (4) performing the correct pipeline behavior to prepare for a control transfer to PAL code (possibly squashing the exception-causing instruction, if any), (5) updating the IPRs at the correct time, and (6) executing the control transfer to PAL code.

The preceding list summarizes the major changes required to the stand-alone simulator to support the OS. Section 2.1.6 relates difficult aspects of this integration. We first describe the software changes to the OS necessary to execute on SMT.

2.1.2 Adapting the OS to execute on SMT

We execute the Compaq/Digital Unix 4.0d[1] operating system, a shared-memory, multiprocessor-aware OS. Modifying it was quite straightforward, because Tru64 Unix is intended to run on conventional shared-memory multiprocessors and is therefore already synchronized for multithreaded operation. By letting SMT appear to the OS as a shared-memory multiprocessor (SMP), required changes to the OS occurred only where the SMT and SMP architectures differed. For Alpha, these differences included *shared* TLB and L1 caches for SMT versus *per-processor* TLB and L1 caches for SMP. Since there were no other changes, only the TLB-related OS code required modification.

The Alpha TLB includes an address space number (ASN) tag on TLB entries. This lets multiple address spaces share the TLB and reduces TLB flushing on context switches. Because multiple threads can *simultaneously* access an SMT processor's shared TLB, manipulating these ASNs requires appropriate mutual exclusion during context switches. We therefore made several changes to the TLB-related code. First, we modified the ASN assignment algorithm to cover multiple executing threads. Second, we replicated, on a per-context basis, the internal processor registers used to modify TLB entries; this removes a race condition and allows multiple contexts to process a TLB miss in parallel. Third, we removed the TLB shootdown code, which is unnecessary in the uniprocessor SMT.

Differences in the architectural interfaces to the caches between an SMT processor and an MP do not necessitate OS modifications. The interface provides commands to flush the L1 instruction and data caches. In an SMT, this causes flushing of the thread-shared cache rather than a thread-local cache. Since the cache is soft state, the extra flushing that results may be unnecessary, but it is never incorrect.

---

1. We fixed a bug in 4.0d that excessively limited the number of netisr threads that can respond to network interrupts. Tru64 Unix 4.0f appears to have addressed this bug.

These modifications constitute the set of minimal changes required to run Digital Unix on an SMT and do not include the numerous opportunities for optimizations. For example, spin routines in the OS, such as the idle loop and the lock-acquire routine, can waste resources on an SMT and could be replaced with SMT-hardware-lock based routines. Another possible optimization would replace the MP OS process scheduler with an SMT-optimized scheduler [70, 59]. We evaluate the impact of removing spinning on SMT in Chapter 4 and plan to investigate other OS optimizations, such as SMT-sensitive thread scheduling, as future work[1]. It is encouraging that an SMP-aware OS can be modified in a straight-forward fashion to work on an SMT processor.

2.1.3 Simulator parameters

This thesis evaluates a variety of SMT and superscalar processor configurations. Table 2.1 lists architectural parameters common to all architectures evaluated. The SMT processor pipeline consists of nine stages, with two stages each dedicated to reading and writing the register file due to its large size. The superscalar processor differs only in its shorter seven-stage pipeline, lacking the need for additional read and write stages.

Our studies focus on CPU and memory performance bottlenecks. In the interest of simulation time, we simulate a zero-latency disk, modeling a machine with a large, fast disk array subsystem. However, all OS code to manipulate the disk is executed, including the disk driver and DMA operations. Modeling a disk-bound machine could alter system behavior, particularly in the cache hierarchy.

2.1.4 Workloads evaluated in subsequent chapters

This thesis examines six different workloads. Each chapter evaluates a subset of these six.

1. SMT introduces a new denial-of-service attack via the TLB. Consider a context that incurs a TLB miss on an instruction A. The processor executes the TLB miss handler to fill the TLB with the entry that instruction A requires and then restarts the miss-causing instruction. However, other contexts may be able to cause instruction A to repeatedly miss in the TLB if they can cause the displacement of the TLB entry *after* the miss handler fills the TLB but *before* instruction A executes again. Incurring a sufficiently high miss rate will continuously displace the entry instruction A requires and prevent that context from making progress. Luckily, today's large TLBs probably prevent a collection of contexts from generating a sufficient frequency of misses.

TABLE 2.1: SMT parameters.

Architectural Parameter	Value
Pipeline	9 stages
Fetch Policy	8 instructions per cycle from up to 2 contexts (the 2.8 ICOUNT scheme of [81])
Functional Units	6 integer (including 4 Load/Store and 2 Synchronization units)
	4 floating point
Instruction Queues	32-entry integer and floating point queues
Renaming Registers	100 integer and 100 floating point
Retirement bandwidth	12 instructions/cycle
TLB	128-entry ITLB and DTLB
Branch Predictor	McFarling-style, hybrid predictor [49]
Local Predictor	4K-entry prediction table indexed by 2K-entry history table
Global Predictor	8K entries, 8K-entry selection table
Branch Target Buffer	1K entries, 4-way set associative
MSHR	32 entries for the L1 caches, 32 entries for the L2 cache
Store Buffer	32 entries
Cache Line Size	64 bytes
I-cache	128KB, 2-way set associative, single port
	2 cycle fill penalty
D-cache	128KB, 2-way set associative, dual ported (only from CPU, r/w). Only 1 request at a time supported from the L2
	2 cycle fill penalty
L2 cache	16MB, direct mapped, 20 cycle latency, fully pipelined (1 access per cycle)
L1-L2 bus	256 bits wide, 2 cycle latency
Memory bus	128 bits wide, 4 cycle latency
Physical Memory	128MB, 90 cycle latency, fully pipelined

The first workload, common to all investigations in this thesis is Apache (version 1.3.4), a popular public-domain Web server run by the majority of Web sites [38]. Because it makes heavy use of OS services[1], it is a rich environment in which to examine OS performance[2]. Most Apache data we present is based on simulations of over one billion instructions, starting when the server is idle. However, the superscalar experiments in Section 3.2.2 were performed on simulations of around 700 million instructions, limited by constraints on simulation time.

We drove Apache using SPECWeb96, a Web server performance benchmark [73]. We configured Apache with 64 server processes and SPECWeb with 128 clients that provide

1. Chapter 3 will show that 75% of execution cycles are spent in the kernel.
2. Apache was designed for portability. Its behavior may not be representative of other web servers, such as Flash [58], which were designed for performance (i.e., high throughput). However, one study comparing Flash and Apache on SMT found little performance difference in a CPU-bound configuration like this dissertation evaluates [60].

requests. To support the request rate needed to saturate Apache, we executed the SPECWeb benchmark as two-driver processes, each with 64 clients. Both Apache and the SPECWeb clients run on separate, networked SimOSs, as we describe in the next section.

Experiments in Chapter 3 evaluate our second workload, a multiprogrammed one composed of all eight applications from the SPECInt95 suite [64], which we simulated for 650 million instructions. SPECInt95 was chosen for two reasons. First, it was commonly used for architecture evaluations, including studies of SMT, and we wished to understand what previous architectural evaluations missed by not including OS activity. Second, the performance characteristics of SPECInt can serve as a baseline to help understand Apache's performance, since Apache is also an integer program.

Finally, Chapter 5 evaluates minithreads on four workloads from the SPLASH-2 benchmark suite [74]. SPLASH-2 is a suite of explicitly parallel scientific applications. We evaluated Barnes, Fmm, Raytrace, and Water-spatial. We adapted each application to SMT by replacing the heavyweight synchronization primitives with the faster SMT hardware lock-based synchronization primitives [83].

2.1.5 Additional tools

Additional tools greatly facilitated our experiments. This section describes three that helped and one that did not prove useful.

Networking workloads, such as Apache, require the interaction of multiple copies of the simulator. We drive Apache with clients from the SPECWeb96 benchmark. One simple simulation methodology executes Apache in the simulator and the SPECWeb96 clients on native machines. The SimOS-alpha distribution lets a simulated machine communicate with real-world computers. However, the 100,000-fold slowdown of the SMT simulator prevents effective communication, because the network code cannot operate properly and TCP will drop messages. Therefore, we built a tool that runs multiple copies of SimOS on a single Alpha. As a result, all machines (i.e., Apache and the SPECWeb96 clients) experience exactly the same slowdown. Packets are generated at a reasonable a rate, and the OS code on both sides of the communication can properly manage the network interface and protocols. Between the SimOS environments, we simulate a direct network connection that transmits packets with no loss and no latency. The simulated network cards inter-

rupt the CPUs every 10 ms, and the network simulator enforces a barrier synchronization across all machines every simulated 10 ms. This barrier keeps the simulators running in lock-step and guarantees deterministic execution of the simulations for repeatability of our experiments.

To allow unobtrusive profiling of the kernel, we ported a version of gprof [28] to execute inside the simulator. Outside of the simulator, gprof works as follows. The compiler compiles a target application with extra instrumentation code that records execution information at procedure call and return points. This extra code collects data during application execution and writes it to a special output file, which the gprof program interprets and displays after execution. Instead of recompiling the OS to include instrumentation code, we implemented a modified version of the instrumentation code directly within the simulator. The code constructs a procedure call stack that is annotated with the information we wish to measure (e.g., cycle counts at all procedure call and exit points). This lets us collect information on kernel execution completely unobtrusively, i.e., *with no effect on workload execution.*

Our implementation differs from standard gprof in one important way: we maintain the complete procedure call stack throughout OS execution. This lets us accurately measure the amount of time spent in every procedure and in all of its descendents. In addition, the call-stack information proved invaluable for debugging both the OS and the simulator, because it provides information on the state of kernel execution at any time. Gprof does not maintain such detailed information on execution state and so cannot quantify the amount of time spent in a procedure and its descendents, for example. The implementors of gprof presumably chose not to maintain a call-stack to limit the overhead and obtrusiveness of the instrumentation code.

The variety of measurements we use to characterize performance made it worthwhile for us to implement a general mechanism to group most statistics by bucket. This lets us differentiate user and kernel behavior, for example. Hardware structures and facilities that support grouping statistics by buckets include the memory system, TLBs, branch prediction hardware, instruction counting code, and most of the statistics that monitor instruction flow through the pipeline. All of the conflict analysis statistics in the branch/cache/TLB hardware also analyze conflicts (and constructive interference) between different buckets.

Our bucket implementation lets a single routine decide, for each statistical event, into which bucket to record the event. This routine provides a simple, centralized way to change how the simulator records statistics. For example, by altering the routine to divide user and kernel events into two buckets, most statistics will record user and kernel behavior separately, and all conflict analysis statistics will report on user/kernel conflicts. However, the simulator's architecture permits the routine to select which bucket based on only limited information about the simulator state[1]. We program these bits to indicate context-specific information, such as user or kernel mode, the currently executing thread id, an interrupt processing flag, and other information.

The TCL interpreter [57] provided by SimOS proved less useful than we had hoped. The simulator invokes the TCL interpreter at the beginning of every cycle, for every instruction committed, and upon every exception. The interpreter lets the simulator be instrumented -- without recompilation -- both to collect statistics and affect simulator behavior, like forcing a checkpoint of simulator state to be recorded. However, its high software overhead made it unacceptable for all but the most simple or temporary operations. In the interest of simulator speed, over time we migrated more functionality from TCL into the simulator proper. We did not remove the TCL interpreter completely, because a few TCL routines modify simulator state at crucial points during the OS boot process.

2.1.6 Difficult problems and lessons learned

This section describes a few of the more complex problems encountered while integrating the application-level SMT processor into SimOS. Many of the issues raised here apply to simulating the operating system on superscalar processors as well. Two factors caused most of the challenges discussed here.

Foremost is the size of the simulator and the complexity of the simulator and operating system. The combined SMT-SimOS simulator consists of 176,000 lines of C code, 8000 lines of TCL code, 12,000 lines of perl scripts, and roughly 200 pages of implementation notes. The core SMT processor simulator alone consists of 49,000 lines of dense C

1. Currently, the routine selects buckets based on 64 bits of simulator state.

code and models the processor at approximately 1 / 100,000 of real time. Compared to the stand-alone, application-level SMT simulator, this represents a 3.3x increase in C code and over a 20x slowdown in simulation speed. For example, the main logical clause that decides if an instruction can issue increased from 24 logical operators to over 80 operators. Compared to the core Gamma version of the SimOS processor simulator, which simulates SMT at the functional level only, the difference grows to 16x in code size and 200x in simulation speed.

Second, the simulator architecture interacts with OS support functionality in complex ways. Specifically, as we discuss below, the triad of speculation, synchronization primitives, and exceptions compound each other, requiring a far more complex implementation than any one alone would necessitate. For example, speculation dramatically increases the types of exceptions that occur.

We begin with a brief description of the simulator architecture (Section 2.1.6.1) to help clarify the issues that follow. Next, we cover the triad of speculation, synchronization primitives, and exceptions (Section 2.1.6.2, Section 2.1.6.3 and Section 2.1.6.4). Finally, we address challenges in debugging the simulator (Section 2.1.6.5) and monitoring OS activity (Section 2.1.6.6). Most of the following discussion centers on the behavior of a single SMT context.

2.1.6.1 A crash course in simulator architecture

Understanding the issues that follow requires a basic understanding of the simulator's architecture. As we mentioned, the SMT processor that we model is based on an out-of-order superscalar pipeline. For each instruction, the simulator internally distinguishes between emulating the instruction (i.e., computing the instruction's result) and simulating instruction flow through the pipeline.

In a real processor, computing an instruction's result occurs mostly in the *execute* stage of the pipeline. For example, a conditional branch resolves in the execute stage; at this time, a misprediction causes the processor to flush the pipeline of newer instructions and resume fetching from the correct target PC. The SimOS-alpha simulator included a partially-implemented out-of-order pipeline simulator that similarly emulated instructions when they reached the execute stage of the pipeline. This approach has two drawbacks.

First, correct program execution requires all issue- and instruction-ordering rules in the simulator to be correct. The implementation complexity of these rules limits the flexibility to experiment with architectural modifications. Second, the results of an instruction are not available to the simulator until the execute stage. Not knowing an instruction's results when it is fetched greatly complicates the gathering of statistics and modeling of perfect branch prediction. Many simulator operations require knowledge of an instruction's results early in the pipeline.

Alternatively, our simulator emulates instructions at *fetch* time. This decision dates from Dean Tullsen's original implementation of the stand-alone, application-level SMT simulator [80]. Figure 2.2 depicts the architecture. For the moment, consider simulating a processor that does not support synchronization. Emulating instructions at fetch time provides the simulator with instruction results at fetch time. With full knowledge of what will happen to an instruction when it flows through the pipeline (i.e., if it will trap or be squashed), gathering statistics and modeling branch prediction become simpler problems. Further, emulating at fetch divorces instruction emulation from simulation of instruction flow through the pipeline. The simulator implementor can model the pipeline and issue rules in as much or as little detail as he/she chooses. Separating instruction emulation from simulation also aids debugging, because bugs in the pipeline model will usually not affect instruction emulation. This helps to isolate bugs.

The presence of synchronization operations complicates instruction emulation at fetch time. Synchronization instructions, loads, and the read-processor-cycle-count (rpcc) instructions are the only instructions whose results depend on when they are emulated. Although a load's result can depend upon when it executes, such rare behavior usually indicates a race condition and probably a bug. Rpcc usually cannot be used for synchronization or communication; hence applications are insensitive to when an rpcc instruction is emulated. Tullsen could have chosen to approximate and emulate synchronization operations at fetch time, like the other instructions, in which case most of the issues in this section would not arise. However, because SMT shares processor resources among many threads, and threads interact through synchronization, modelling synchronization behavior accurately in SMT critically affects overall simulation accuracy.

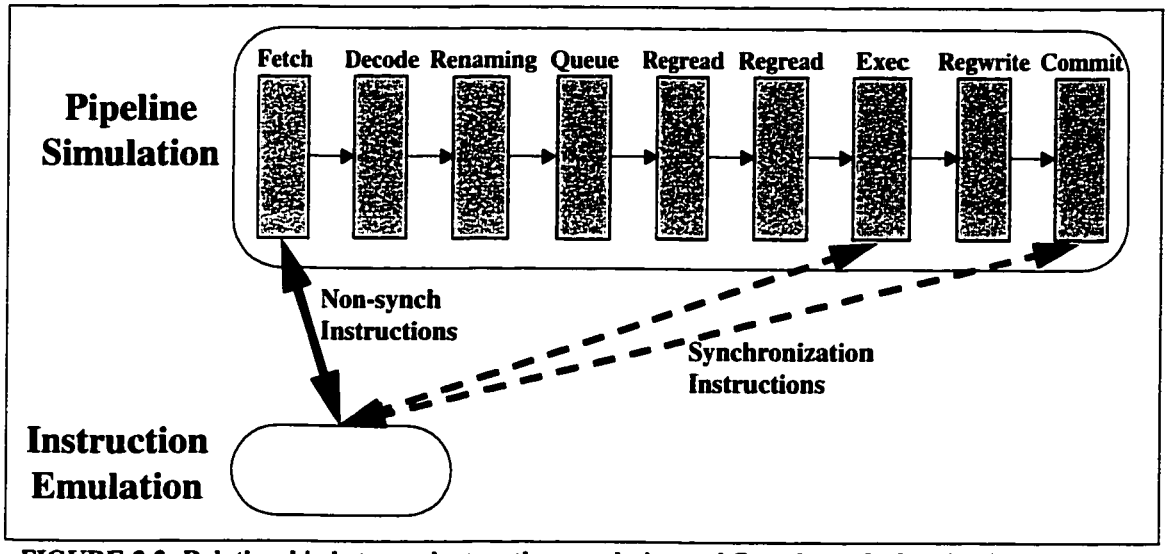


FIGURE 2.2: Relationship between instruction emulation and flow through the pipeline. The simulator emulates most instructions when it fetches them. For accuracy, it emulates synchronization operations when they reach the execute or commit stage.

The simulator models synchronization as follows. At fetch time it approximates the results of a synchronization operation and enters a special "speculative" mode. During this time, it prevents all more recent instructions from causing side-effects and postpones all exceptions. When the synchronization operation executes, the simulator computes the correct result of the operation and *reemulates all newer instructions*. The special 'speculative' mode ends after this reemulation if there are no newer synchronization operations in the pipeline. During the reemulation, the results of any instruction dependent on the synchronization instruction, including conditional branches, can change.

2.1.6.2 Speculation

Speculation increases the frequency and types of illegal instructions. Correctly written applications and the kernel usually contain only well-formed and legal instructions. However, when speculating on the wrong-path, instructions can quickly deteriorate into illegality. For example, an incorrectly predicted indirect jump could direct the processor to fetch an address that does not contain valid code. The simulator writer must define a behavior for each illegal instruction. On an application-level simulator, with its simplistic notion of protection and no support for exceptions, fewer operations are illegal, and the handling of

each illegal operation is simple. It usually involves just halting the simulator when the operation becomes non-speculative.

Including the operating system in simulation complicates speculation. The number of illegal behaviors increases and the correct handling of such operations becomes more complex. It may have been possible to simplify the issue by causing all illegal operations to generate a general instruction fault trap. However, in the interests of debugging, we would like each illegal operation to cause reasonable error behavior. For example, virtual memory violations should cause a VM-related trap. Modelling reasonable error behavior also improves simulator accuracy, because wrong-path instructions and their results compete for resources with correct-path instructions. We devoted much implementation effort to defining reasonable behavior for most failure cases in instruction semantics, because they all occurred at one point or another. For example, what should happen when a PAL instruction in the TLB miss flow that bypasses all memory system checks attempts to write an address that does not exist on the machine?

Operations that manipulate privileged hardware state, such as special instructions and exceptions, posed an additional challenge with respect to speculation. Most privileged hardware state does not support speculative updates[1]. Therefore, we had to implement speculative 'dummy' versions of all operations that affect that state. Dummy versions perform identically to the real ones, except they do not modify privileged hardware state. We call the dummy versions during speculative execution or following synchronization primitives, as described in Section 2.1.6.1. The simulator must then determine whether to execute the real or dummy version, sometimes a difficult decision.

2.1.6.3 Synchronization

To understand the interaction of synchronization and simulation, recall first that the simulator emulates most instructions at fetch time. At this time, if possible, it determines the instruction's result and also whether the instruction will eventually be squashed or not, that is, it's speculative nature. Synchronization operations complicate the determination of

1. We tried implementing speculative exceptions, but it slowed down the simulator substantially and did not greatly affect SMT processor performance.

subsequent instructions' speculative nature. Note that this uncertainty differs from the uncertainty associated with traditional speculation.

Two kinds of uncertainty exist, each at a different level. The first kind of uncertainty occurs when an instruction executes before the *processor* has determined that all previous instructions will not cause an exception. This is traditional speculation. The second kind of uncertainty occurs when the *simulator* cannot determine the speculative nature of an instruction at fetch time (when it emulates the instruction) because an older synchronization instruction in the pipeline has not resolved. The synchronization instruction's result (e.g., whether a lock was successfully acquired or not) can alter the results of any newer instructions that follow. In the worst case, data dependencies may keep the simulator from knowing the result of an instruction until it actually issues. It may not determine if the instruction is on the wrong path until long after it reaches the commit stage and awaits graduation. In other words, statistics we wish to gather for only correct-path instructions potentially must be deferred until the commit stage. This complicates statistics gathering, especially OS-related statistics, such as profiling information. Further, it partly explains limits on the statistic bucket mechanism's ability to determine buckets based on only limited information about simulator state.

Deferring counter updating until the relevant instruction graduates guarantees counting only correct-path instructions. However, it carries a price in complexity. When the event to be recorded occurs, the simulator must make a note. It converts that note to an update of the relevant statistic when the instructions surrounding the event commit. The simulator dedicates much code to buffering these events by associating them with a particular instruction. This buffering becomes even more involved for events that lack an associated instruction. For example, upon a DTLB miss, the simulator squashes the offending instruction.

In a few cases, uncertainty about instructions in the shadow of an unresolved synchronization operation caused the simulator to defer OS events until the synchronization was resolved. This solution exacts a cost in simulator accuracy: we are altering the simulated machine due to simulator implementation issues. For example, we postponed traps until all synchronization operations were resolved, which may have occurred after the trap became non-speculative. A real machine would trap as soon as the trap's cause became

non-speculative. However, our statistics indicate that this distortion in simulator behavior rarely occurs[1].

2.1.6.4 Exceptions

Because exceptions by nature occur infrequently, bugs in exception handling code tended to be subtle and involved. For example, the simulator must behave correctly when multiple exceptions occur in a single cycle (within a single context). It can only trap to begin processing of at most one exception per cycle and so must choose the correct one. Correctness requires a prioritization of exceptions. SimOS-alpha ordered exception priority, but a few painful bugs proved the order to be incorrect.

Interrupts constitute the only non-synchronous exception. The synchronization and speculation issues described previously make it difficult to determine when the simulator may safely trap an interrupt or squash an instruction. In addition, when an interrupt occurs in the same cycle as other exceptions, and it is trapped, the simulator must choose the correct PC at which to resume when interrupt processing completes.

2.1.6.5 Debugging the simulator

Debugging the simulator demanded tremendous effort because of its size, complexity, and large number of interacting components. For example, consider the nasty class of bugs that cause the simulated kernel to panic after a large number (more than 100 million) of instructions. Possible bug sources include the simulated OS, instruction emulation code, TCL routines, the pipeline simulator, and so on. In addition to not knowing what part of the simulation system might be causing the problem, we also do not know when the bug occurred. The curse of long delays between bug occurrence and detection is infamous in many other domains as well, such as debugging cache coherence protocols.

The Gamma processor simulator proved invaluable in tracking down simulation bugs in long running simulations. Gamma simulates SMT at the functional level, only emulating instructions. A powerful but time consuming debugging technique consists of running the SMT simulator until the bug manifests itself, then running the Gamma simulator for

1. Surprisingly, our statistics indicate that in no simulation did synchronization delay processing of a trap. However, we cannot simplify because we cannot guarantee that the event will not occur.

roughly an equal number of instructions. We program each simulator to print out every instruction executed and look for the first difference between the two instruction lists. This technique requires the Gamma simulator to emulate instructions in exactly the same order as the SMT simulator. We must also verify that each instruction in both simulators produces the same result, including operations on the processor cycle counter. The large size of the instruction traces, frequently over 1GB, and the effort required to filter out all differences besides bugs makes this technique applicable only for the most subtle bugs.

The key to debugging the simulator, then, was to cause bugs to surface as early as possible. We hope to notice bugs within the same simulated cycle in which they occur, and preferably within the same pipeline stage[1]. A bug may manifest itself as a warning message or other abnormality in the simulation log file. However, as the log files usually exceed 100MB, a single-line warning message can get lost in the mountain of data, especially if the bug does not cause catastrophic failure. Instead of generating a warning message, the bug should cause the simulator to halt with an error. To this end, we freely added ASSERT statements to the simulator during development to assert invariants about execution. The 45,000 lines of C code in the core SMT processor simulator contain 2,600 ASSERT statements that each check multiple invariants. The ASSERT statements cost roughly 2x in simulator speed. However, the huge benefit of isolating a bug soon after it occurred far outweighed this slowdown.

Underlying all our debugging techniques is a deterministic simulator. Multiple simulations starting from the same state must yield identical results to ensure that all bugs are reproducible. A single SMT simulator executes as a single process and does not interact with the real world, so naturally executes deterministically. However, ensuring deterministic execution among multiple SimOS's communicating via the network simulator required careful ordering of messages with respect to the barrier synchronization and checkpointing facility.

1. A cycle of simulation includes simulating each pipeline stage separately.

2.1.6.6 Monitoring OS activity

In the simulator we wish to keep statistics about many different OS activities. Examples include context switches, the creation and death of threads, time spent in kernel mode and user mode, and so on. Measuring these events is complicated by three factors.

First, as mentioned previously, the presence of speculation and synchronization means that accurate information about an event might not be determined until instructions around the event graduate. While some information can be propagated with an instruction, this may not be feasible for every event. Furthermore, since instruction emulation occurs at fetch time, there is an information gap between the state of the machine with respect to an instruction that has just committed and the current simulator state, i.e., the state with respect to the most recently fetched instruction or trap. For example, upon a context switch, how does the simulator determine the thread to which the processor switched? Suppose that the simulator detects context switches by the branch instruction that returns from the context switch routine. When the branch instruction reaches the commit stage, it knows that a context switch occurred. To determine the destination thread of the switch, it may try to read the OS memory word that specifies the processor control block (pcbb) of the current thread. However, that memory word is updated at instruction emulation (fetch) time. It therefore reflects the state of the machine after the most recent instruction emulation, which may differ from the state when the branch instruction was emulated. Choosing the correct source of information for each routine in the simulator code demanded great care.

Second, since we model a real TLB and support real physical addresses, performing virtual-to-physical translation can present a challenge. The simulator translates most virtual addresses into physical addresses as part of instruction emulation and notes the physical addresses for later reference. However, the simulator occasionally needs to access memory for which it has only a virtual address. Translating the address is straightforward if it resides in the TLB, but the simulator cannot rely on that. Therefore, we implemented a TLB miss handler inside the simulator to walk the page table. This is an inadequate long-term solution, since the simulator can race the operating system for access to the page

table. Although we have not encountered this situation, it is possible that the simulator will crash if it attempts to walk the page table while the OS is updating it.

The presence of exceptions constitutes the final complicating factor. Exceptions mark important transition points between time periods we wish to measure, such as a user/kernel transition or entering/exiting an interrupt handler; however, they consist of many separate steps. Defining the step at which the transition occurs and recording the transition can be challenging. For some events, such as the PAL call that initiates a system call, we can define the transition point simply as the cycle in which the call_PAL instruction reaches the decode stage, at which point the processor recognizes the call and prepares to trap. The simulator annotates the instruction with the cycle at which it reached the decode stage and logs the transition in its statistics when the instruction graduates. (We must wait until it graduates for reasons discussed above.) Other events are more complicated. Consider a DTLB miss in an instruction that itself caused an ITLB miss that has completed. Only two instructions surrounding the end of the ITLB miss and the beginning of the DTLB miss will reach the commit stage: the REI instruction that returns from the ITLB miss handler, and the first PAL instruction of the DTLB miss handler; the double miss causing instruction itself is squashed. The REI instruction marks the end of the ITLB miss handler, but we lack an instruction that carries information about the DTLB miss. Further, we lack an instruction that indicates time spent in user space. So we must annotate the first instruction of the DTLB miss handler with information about time spent in user space fetching the DTLB miss-causing instruction, when the instruction trapped, when the miss handler instruction was fetched, etc.. Decoding this information at commit time contributed to the 33x blowup in commit stage code size (from 150 to 4800 lines).

2.1.7 Summary of infrastructure

We integrated SimOS and an SMT application-level simulator to produce a detailed simulator infrastructure capable of modeling every instruction that would execute on a real machine, including operating system code. Developing an SMT-capable OS to execute on this simulator involved only minor modifications to a multiprocessor-aware OS, all of which occurred in TLB-miss code. Developing and using this simulator presented many

challenges, primarily due to simulator size and a triad of compounding factors: speculation, synchronization, and exceptions.

2.2 Measurement methodology issues

Over the course of upgrading the application-level simulator, implementing OS support, and continually examining simulation results over a variety of workloads, we have come to a few insights that proved crucial to understanding simulation results. This section covers two such insights, beginning with an observation about branch prediction, and ending with a discussion of cycle counting methodology.

2.2.1 Modeling perfect branch prediction is impossible!

In general, whenever a simulator predicts future processor events in a way that can alter those events, perfect prediction is impossible. The following example shows how this feedback can occur in the case of branch prediction. We informally argue that for any execution-driven SMT simulator of an ISA that contains synchronization instructions resolved in the execute stage, modeling perfect branch prediction is impossible. We also explain why this holds for many similar architectures.

Consider a multithreaded or SMT simulator that attempts to model perfect branch prediction. Suppose that in a particular cycle the simulated processor fetches for context A a non-blocking, lock-acquire synchronization instruction (e.g., store-conditional), followed by a conditional branch instruction that depends on the result of the lock-acquire. Soon after, perhaps in the same cycle, context B fetches a lock-release instruction for the same lock. Both the lock-acquire and lock-release instructions flow through the pipeline and execute within a few cycles of each other. If the lock-release executes before the acquire, the acquire will succeed; otherwise, it will fail. In the cycle in which the processor fetches the lock-acquire instruction, the processor must decide whether the instruction will succeed in order to determine the correct branch-target PC from which to fetch the next cycle. However, at this point the lock-acquire operation has not yet executed.

The simulator cannot guarantee branch prediction success because of a feedback loop between the prediction and the future result of the lock-acquire. The success of the lock-

acquire can depend on the result of the prediction and also influences it. Predicting the target PC alters many components of the pipeline. Through synchronization operations, these components can in turn affect the branch's outcome. We present an example of how this feedback may occur.

Without loss of generality, suppose that the simulator determines that the synchronization operation will succeed - i.e., that the lock-release will execute before the lock-acquire. (The simulator could determine this by temporarily rolling the simulation forward to see if the lock-acquire operation succeeds, then unrolling, for example.) Based on this determination, the simulator computes the target PC of the branch and initiates an I-cache access for it. Suppose further that the I-cache access misses. This causes a bus transaction to the L2 cache. The L2 misses and the resulting fill from main memory then displaces a data cache line. A load instruction in context B delays because it misses in the L2 when trying to access that expelled data cache line. This delay consequently delays the execution of context B's lock-release, which is dependant on the load, until *after* the lock-acquire operation of context A executes. These events result in the failure of the lock-acquire operation and misprediction of the conditional branch. Conversely, if the processor had determined that the synchronization operation would fail and fetched from the other branch-target PC, the I-cache miss might not have happened. This would result in early execution of the lock-release, success of the lock-acquire, and again a misprediction. Regardless of the PC from which the simulator fetches, it will always mispredict the branch in this scenario!

When predicting instruction results (such as predicting branches), the impossibility of perfect prediction depends on the existence of an instruction whose result depends on timing. This is because modifications to the processor pipeline that we wish to model, such as perfect branch prediction, will only affect the timing of instruction execution, not instruction semantics. In the preceding example, synchronization fills this role. However, other timing-dependent operations, such as rpcc or unsynchronized communication through shared memory could similarly thwart perfect prediction attempts.

Most simulators that model a pipeline in enough detail to provide a feedback loop between predicting and affecting simulation can suffer from this phenomenon. Specifically, designing a simulator to emulate instructions in the execute stage does not avoid this

problem, and in fact, may worsen it. Such a simulator still must predict a branch at fetch time, before the branch instruction executes. The window between when the instruction is fetched and when it executes provides an opportunity for feedback to result in a misprediction. Even in the absence of feedback or synchronization, such a simulator may have more difficulty modeling perfect branch prediction than a simulator that emulates instructions at fetch time, because the simulator does not know the results of instructions until they execute.

Luckily, mispredicted branches due to this behavior occurred infrequently in our simulations. Our model of an ideal predictor computes the results of branches in the fetch stage. These results guide the choice of the fetch PC. Measurements indicate that the percentage of instructions squashed due to branch mispredictions when modeling an ideal predictor does not exceed 0.5% of all instructions fetched.

2.2.2 Measuring time on modern processors

We finish this chapter with a brief discussion of a cycle-counting methodology. We begin with single-threaded processors, and then cover SMT-specific issues.

2.2.2.1 Single-threaded processors

In this thesis we attribute cycles to different activities such as spinning or executing kernel code. Different methods for measuring the length of these activities include measuring at the fetch stage, the commit stage, or some combination of the two. For example, we can define the time taken by the lock-acquire routine to be the time between when the processor commits the instruction that calls the lock-acquire routine and when it commits the return instruction at the end of the routine.

Unfortunately, pipelined processors resist accurate measurement of any particular activity because instructions from different activities coexist in the pipeline. This problem occurs even on an *in-order processor with a single functional unit*! Continuing the above example, an I-cache miss incurred by the first instruction of the lock-acquire routine will not count as part of the time attributed to the lock-acquire routine. In the interest of brevity, we omit the discussion of other, similar measurement methodologies, all of which suffer from similar measurement difficulties. Superscalar and out-of-order pipelines further

complicate time measurement because they increase the potential overlap and interaction of instructions from different activities.

We define activities as beginning and ending when the processor fetches the relevant instruction. However, we record the measurement only when the instruction commits. This avoids misspeculated instructions and also problems with uncertainty and synchronization described in Section 2.1.6.3. We measure exceptions as beginning on the cycle in which the processor indicates an exception. We mark the exception handler as completing when the final branch (REI) instruction resolves.

Note that for an activity of sufficient time duration, these time measurement issues disappear. The issues arise at both the beginning and ending boundaries of an activity due to overlap with instructions from the preceding and succeeding activities, respectively, in the pipeline. Each pipeline has a maximum amount of time an instruction can spend passing through it. The maximum time is determined by the pipeline length, maximum memory hierarchy delay and other factors. As the length of the activity becomes much longer than the maximum instruction time in the pipeline, the maximum possible overlap becomes insignificant relative to the length of the activity.

2.2.2.2 SMT processors

SMT adds another level of complexity to time measurement, independent of the pipeline stage at which measurements are performed, for two reasons. First, multiple threads execute on the processor simultaneously. On single-threaded processors, the overlap between consecutive activities, which distorts all time measurement methods, occurs only at the beginning and end of the activity. On SMT, inter-activity interactions can occur every cycle. Tolerating latency necessarily mixes instructions from different activities on the processor. Therefore, the more effectively a processor tolerates latency, the greater interference we can expect in any technique that tries to measure a single activity's time. We know of no time measurement method that can factor out inter-activity interactions on SMT.

Second, an activity can execute on SMT on more than one context simultaneously. On a superscalar, counting elapsed cycles sufficed because only one activity executes at a

time. On SMT we also wish to capture what *fraction* of contexts an activity occupies as it executes.

We measure time on SMT by counting the fraction of context-cycles taken by an activity. Consider the execution in Figure 2.3, in which two contexts on a four-context SMT execute an activity in overlapping intervals during an eight-cycle simulation. Activity A executes five cycles on context 0 and five cycles on context 1. We calculate that it occupied 31% of total time (10 context-cycles / 32 overall context-cycles).

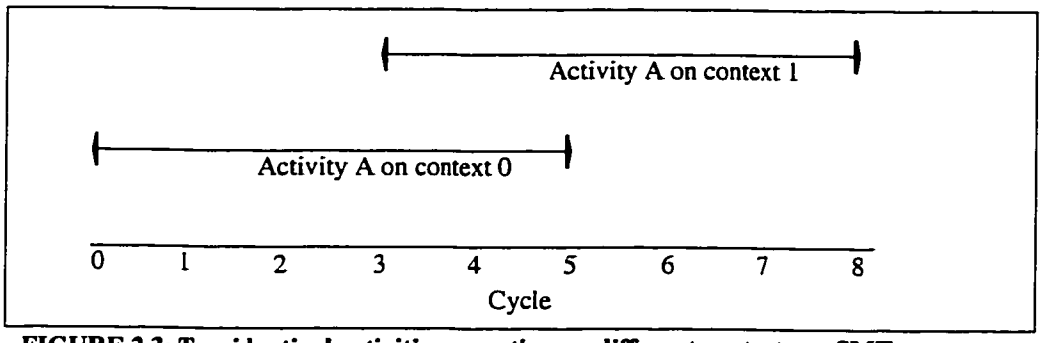


FIGURE 2.3: Two identical activities executing on different contexts on SMT.

2.3 Chapter summary

This chapter examined our experimental methodology. Our simulation infrastructure integrates an SMT application-level simulator into the SimOS machine simulation framework. We described many aspects of the integration process, OS issues, integration challenges and measurement tools. We also discussed two methodological issues critical to understanding experimental results. First, we showed that feedback between pipeline stages prevents modeling of perfect branch prediction. Second, we explored the difficulty of precisely measuring time intervals on modern processors. The interaction of multiple instructions simultaneously resident on the processor prevents a clean separation between consecutive intervals. Consequently, no methodology can accurately capture short intervals, and most measurement methodologies suffice for long intervals.

Chapter 3

An analysis of operating system behavior on SMT

As a general-purpose throughput-enhancing mechanism, SMT is especially well suited to applications that are inherently multithreaded, such as database and Web servers, as well as multiprogrammed and parallel scientific workloads. This chapter provides the first examination of (1) operating system behavior on an SMT architecture, and (2) a Web server SMT application. For server-based environments, the operating system is a crucial component of the workload. Previous research suggests that database systems spend 30% to 40% of their execution time in the kernel [9], and our measurements show that the Apache Web server spends over 75% of its time in the kernel. Therefore any analysis of their behavior should include operating system activity.

Operating systems are known to be more demanding on the processor than typical user code for several reasons. First, operating systems are huge programs that can overwhelm the cache and TLB due to code and data size. Second, operating systems may impact branch prediction performance, because of frequent branches and infrequent loops. Third, OS execution is often brief and intermittent, invoked by interrupts, exceptions, or system calls, and can cause the replacement of useful cache, TLB and branch prediction state for little or no benefit. Fourth, the OS may perform spin-waiting, explicit cache/TLB invalidation, and other operations not common in user-mode code. For these reasons, ignoring the operating system (as is typically done in architectural simulations) may result in a misleading characterization of system-level performance. Even for applications that are not OS-intensive, the performance impact of the OS may be disproportionately large compared to the number of instructions the OS executes.

As the first study of OS behavior in an SMT environment, our goal is to answer several basic questions. First, how would previously reported results change, if at all, when the operating system is added to the workload? In particular, we wish to verify the IPC

results of previous studies to see whether they were overly optimistic by excluding the OS. For these studies, we used a multiprogrammed workload consisting of multiple SPECInt benchmarks. Second, and more important, what are the key behavioral differences at the architectural level between an operating-system-intensive workload and a traditional (low-OS) workload, both executing on SMT? For example, how does the operating system change resource utilization at the micro-architecture level, and what special problems does it cause, if any, for a processor with fine-grained resource sharing like SMT? For this question, we studied one OS-intensive application, the widely-used Apache web server [38]. We compared the Apache workload and the SPECInt workload to study the differences in high-OS and low-OS usage. Third, how does a Web server like Apache benefit from SMT, and where does it spends its time from a software point of view? This analysis is interesting in its own right, because of the increasing importance of Web servers and similar applications. We therefore present results for Apache on an out-of-order superscalar as well as SMT. Overall, our results characterize both the architectural behavior of an OS-intensive workload and the software behavior (within the OS) of a key application, the Apache Web server.

The chapter is organized as follows. Section 3.1 details our measurement methodology, our simulation environment, and the workloads we use. Section 3.2 presents measurement results for our two workloads on SMT including operating system execution. The first half of Section 3.2 examines a multiprogrammed workload consisting of SPECInt applications, while the second half focuses on the Apache workload. Section 3.3 describes previous work and its relationship to our study, and we conclude in Section 3.4.

3.1 Methodology

The experiments in this chapter follow the methodology described in Chapter 2. They focus on two workloads: a multiprogramming workload of SPECInt95 benchmarks and the Apache Web server, described in that chapter. To more precisely characterize the impact of the OS on performance, we compare the simulation of a workload that includes the OS with one that models only application code. The application-only simulations are performed with the application-level simulator, described in Section 2.1.1.2.

3.2 Results

This section presents results from our SimOS-based measurements of operating system behavior and its impact on an SMT processor. In Section 3.2.1 we consider a SPECInt multiprogrammed workload; Section 3.2.2 examines an Apache workload and compares it to results seen for SPECInt.

3.2.1 Evaluation of SPECInt workloads

Traditionally, architects have based processor and memory subsystem design decisions on analyses of scientific and program development workloads, as typified by the SPECInt benchmark suite. However, most such analyses examine user-mode code only. In this section we evaluate the appropriateness of that methodological strategy in the context of simultaneous multithreading. We wish to answer two questions in particular. First, what is the impact of including (or excluding) the operating system on SMT, even for a multiprogrammed workload of SPECInt benchmarks? While we expect OS usage in SPECInt to be low, previous studies have shown that ignoring kernel code, even in such low-OS environments, can lead to a poor estimation of memory system behavior [29, 1]. Second, how does the impact of OS code on an eight-context SMT compare with that of an out-of-order superscalar? SMT is unique in that it executes kernel-mode and user-mode instructions *simultaneously*. That is, in a *single* cycle, instructions from *multiple* kernel routines can execute along with instructions from *multiple* user applications, while *all* are sharing a single memory hierarchy. In contrast, a superscalar alternates long streams of user instructions from a single application with long streams of kernel instructions from a single kernel service. This difference may impact memory system performance differently in the two architectures. In Section 3.2.2, we examine similar questions in light of Apache, a more OS-intensive workload.

3.2.1.1 The OS behavior of a traditional SPEC integer workload executing on an SMT processor

Figure 3.1 shows the percentage of execution cycles for the multiprogrammed SPECInt95 benchmarks that are spent in user space, kernel space, or are idle when executing on an SMT processor. During program start-up, shown to the left of the dotted line, the operating

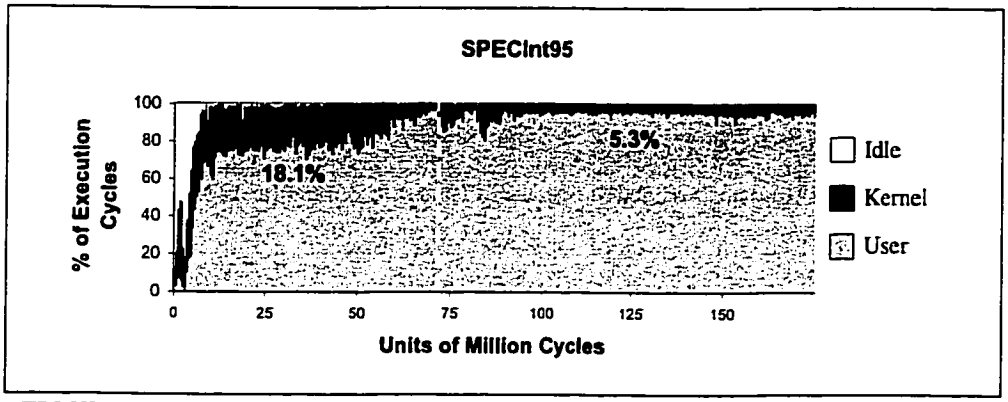


FIGURE 3.1: Breakdown of execution cycles when SPECInt95 executes on an SMT. Cycles spent in the kernel as a percentage of all execution cycles are shown as the dark color at the top.

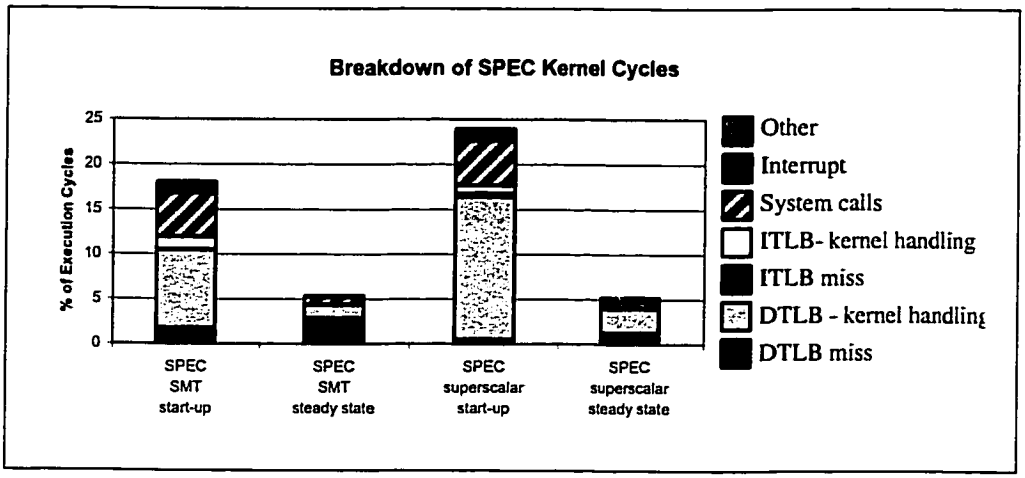


FIGURE 3.2: Breakdown of kernel time for SPECInt95.

systems presence is 18% of execution cycles on average. Once steady state is reached, it drops to a fairly consistent 5%, which is maintained at least 1.6 billion cycles into execution (only a portion of that is shown in the figure). The higher OS activity during program initialization is primarily due to TLB miss handling (12% of all execution cycles) and system calls (5%), as shown in Figure 3.2. Most of the TLB activity centers on handling data TLB misses in user space (roughly 95%). The TLB misses result in calls to kernel memory management, and page allocation accounts for the majority of these calls, as shown in Figure 3.3. The majority of application-initiated system calls (Figure 3.4) are for the file system; in particular, reading input files contributes 3.5% to execution cycles, which is consistent with applications reading in source and/or configuration files. Process creation and control and the kernel preamble (identifying and dispatching to a particular system

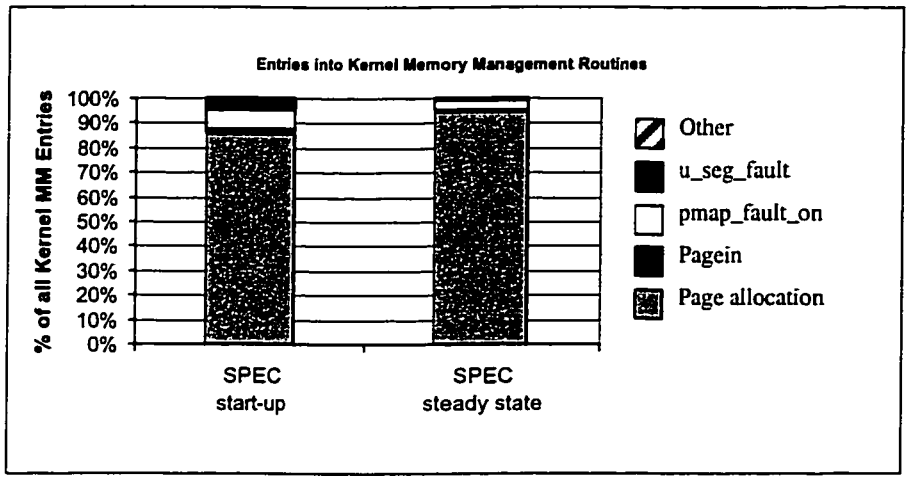


FIGURE 3.3: Incursions into kernel memory management code by number of entries.

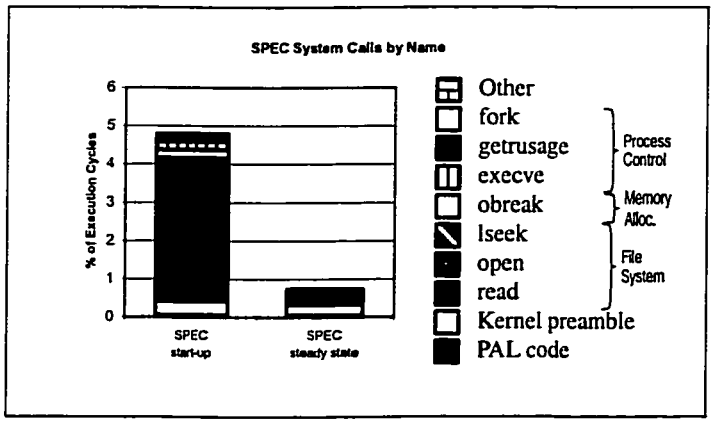


FIGURE 3.4: System calls as a percentage of total execution cycles.

call) fill most of the remaining system call time. Note that kernel activity dwarfs the execution of Alpha PAL code.

Once steady state is reached, kernel activity falls to 5% of execution cycles, but keeps roughly the same proportion of TLB handling and system call time as during start-up. The only significant change is a reduction in file read calls, since program execution has shifted away from initialization.

Table 3.1 shows the distribution of instructions across the major instruction categories in the kernel; these values are typical for integer applications, including the SPEC integer benchmarks. Kernel instructions differ from user instructions in three respects. First, roughly half of the memory operations in program start-up, and one-third of loads and

two-thirds of stores in steady state, do not use the TLB, i.e., they specify physical addresses directly. Second, kernel control transfers include PAL entry/return branches. Third, compared to user code, kernel code in steady state has half the rate of conditional branches taken. However, since the kernel executes a small portion of the time, the overall impact of these differences is small.

TABLE 3.1: Percentage of dynamic instructions in the SPECInt workload by instruction type. The percentages in parenthesis for memory operations represent the proportion of loads and stores that are to physical addresses. A percentage breakdown of branch instructions is also included. For conditional branches, the number in parenthesis represents the percentage of conditional branches that are taken.

	Program Start-up			Steady State		
Instruction Type	**User**	**Kernel**	**Overall**	**User**	**Kernel**	**Overall**
Load	19.5	16.5 (51%)	19.2 (5%)	20.0	12.2 (35%)	19.7 (1%)
Store	12.3	19.0 (57%)	13.1 (10%)	9.6	11.8 (68%)	9.7 (3%)
Branch	15.1	15.9	15.3	14.8	15.0	14.9
Conditional	(64%) 65.9	(56%) 65.3	(63%) 65.8	(56%) 68.3	(26%) 59.9	(54%) 68.0
Unconditional	19.5	14.1	18.8	18.3	6.5	17.8
Indirect Jump	14.7	11.7	14.3	13.3	5.5	13.0
PAL call/return	.01	8.9	1.1	.01	28.1	1.2
Total	100.0	100.0	100.0	100.0	100.0	100.0
Remaining Integer	50.0	48.6	49.7	53.3	61.0	53.5
Floating Point	3.1	0.0	2.7	2.3	0.0	2.2

3.2.1.2 What do we miss by not simulating the operating system on SPECInt workloads?

Table 3.2 (top part) shows the total miss rate in several hardware data structures when simulating both SPECInt95 and the operating system on an SMT. The total miss results mirror what other researchers have found in single-threaded processor studies, namely, that the operating system exhibits poorer performance than SPECInt-like applications [1, 29]. The kernel miss rate in the branch target buffer is particularly high, because of two factors: the OS executes so infrequently that it cannot build up a persistent branch target state, and most kernel misses (78%) displace other kernel entries or are mispredictions due to repeated changes in the target address of indirect jumps.

The miss-distribution results in the lower part of Table 3.2 indicate that, with the exception of the instruction cache, conflicts within or between *application* threads were responsible for the vast majority of misses. Kernel-induced conflict misses accounted for

TABLE 3.2: Total miss rate and distribution of misses in several hardware data structures when simulating both SPECInt95 and the operating system on SMT. The miss categories are percentages of all user and kernel misses. Bold entries signify kernel-induced interference. User-kernel conflicts are misses in which the user thread conflicted with some type of kernel activity (the kernel executing on behalf of this user thread, some other user thread, a kernel thread, or an interrupt).

	Miss Percentages									
	Branch Target Buffer		**L1 Instruction Cache**		**L1 Data Cache**		**L2 Cache**		**Data TLB**	
	User	**Kern**	**User**	**Kern**	**User**	**Kern**	**User**	**Kern**	**User**	**Kern**
Total miss rate	30.5	75.2	1.8	8.4	3.2	18.8	0.9	10.5	0.5	3.2
Cause of misses	**Percentage of Misses Due to Conflicts (sums to 100%)**									
Intra-thread conflicts	51.0	**4.9**	6.5	**.2**	14.6	**.8**	34.7	**1.2**	17.5	**.2**
Inter-thread conflicts	39.5	**1.1**	33.7	**.2**	59.5	**5.2**	18.9	**.7**	64.5	**.8**
User-kernel conflicts	**1.8**	**1.7**	**4.6**	**3.2**	**5.7**	**6.6**	**6.2**	**.9**	**8.2**	**8.8**
Invalidation by OS			**40.9**	**10.7**						
Compulsory			.01	.01	7.3	.3	.5	36.6		

only 10% of BTB misses, 18% of data cache misses, 9% of L2 cache misses and 18% of data TLB misses. In contrast, the majority of instruction cache misses (60%) were caused by the kernel. Compulsory misses are minuscule for all hardware structures, with the exception of the L2 cache, in which the kernel prefetches data for the applications and therefore absorbs the cost of many first reference misses for both.

At a high level, the kernel's poor hardware-component-specific performance is ameliorated by the infrequency of kernel execution for the multiprogrammed SPECInt workload. Table 3.3 (columns 2 through 4) illustrates this effect by comparing several architectural metrics for the SPECInt workload executing in steady state on an SMT, with and without operating system activity. The numbers indicate that instruction throughput dropped only slightly due to the OS (5%) and, with few exceptions, the utilization of the thread-shared hardware resources moderately degraded when including the kernel. Those hardware components in which we observe a large percentage drop in performance did not greatly affect the performance bottom line, because they had not exhibited particularly bad behavior originally.

The rise in the number of speculative instructions squashed was the most serious of the changes caused by simulating the kernel and depended on the interaction between two

TABLE 3.3: Architectural metrics for SPECInt95 with and without the operating system for both SMT and the superscalar. The maximum issue for integer programs is 6 instructions on the 8-wide SMT, because there are only 6 integer units.

Metric	SMT			Superscalar		
	SPEC only	SPEC+OS	Change	SPEC only	SPEC+OS	Change
IPC	5.9	5.6	-5%	3.0	2.6	-15%
Average # fetchable contexts	7.7	7.1	-8%	1.0	0.8	-20%
Branch misprediction rate (%)	8.1	9.3	15%	5.1	5.0	-2%
Instructions squashed (% of instructions fetched)	15.1	18.2	21%	31.8	32.3	2%
L1 I-cache miss rate (%)	1.0	2.0	90%	0.1	1.3	1200%
L1 D-cache miss rate (%)	3.2	3.6	15%	0.6	0.5	-15%
L2 miss rate (%)	1.1	1.4	27%	1.0	1.8	72%
ITLB miss rate (%)	0.0	0.0		0.0	0.0	
DTLB miss rate (%)	0.4	0.6	36%	0.04	0.05	25%

portions of the fetch engine, the branch prediction hardware and the instruction cache. Branch mispredictions rose by 15% and instruction cache misses increased 1.9 times, largely due to interference from kernel execution. Instruction misses were induced primarily by cache flushing that was caused by instruction page remapping, rather than by conflicts for particular cache locations. The rise in instruction misses caused, in turn, an 8% decrease in the number of fetchable contexts (i.e., those contexts not servicing an instruction miss or an interrupt). Because simulating the kernel reduced the average number of fetchable contexts, a mispredicting context was chosen for fetching more often and consequently more wrong-path instructions were fetched.

Surprisingly, the kernel has better branch prediction than the SPECInt applications, despite its lack of loop-based code. (When executing the two together, the misprediction rate in the user code is 9.3 and in the kernel code is 8.2 (data not shown)). Most conditional branches in the kernel are used in diamond-shaped control in which the target code executes an exceptional condition. Although the kernel BTB miss rate is high, the default prediction on a miss executes the fall-through code, and therefore more kernel predictions tend to be correct.

In summary, despite high kernel memory subsystem and branch prediction miss rates, SMT instruction throughput was perturbed only slightly, since kernel activity in SPECInt

programs is small and SMT hides latencies well. Therefore researchers interested in SMT bottomline performance for SPECInt-like scientific applications can confidently rely on application-level simulations. However, if one is focusing on the design of a particular hardware component, such as the data TLB, or a particular hardware policy, such as when to fetch speculatively, including the execution-time effects of the operating system is important.

3.2.1.3 Should we simulate the operating system when evaluating wide-issue superscalars?

In terms of overall execution cycles, the operating system behaves similarly on both an out-of-order superscalar and an SMT processor when executing the SPECInt benchmarks. A superscalar processor spends only a slightly larger portion of its execution cycles during start-up in the OS compared to SMT (24% versus 18% (data not shown)). The percentage of operating system cycles in steady state is the same for both processors.

Likewise, the distribution of OS cycles in both start-up and steady state is similar on the superscalar and the SMT processor (shown in Figure 3.2). One exception is the larger portion of time spent by the superscalar on kernel miss handling for the data TLB. Also, kernel processing of DTLB misses exhibits poor instruction cache behavior, which inflates the time spent in this code. The kernel instruction cache miss rate on the superscalar is 13.8% (compared to a minuscule 0.3% for user code) and 81% of these misses are caused by kernel DTLB miss-handling code.

At the microarchitectural level, the operating system plays a different role on an out-of-order superscalar. Instruction throughput on the superscalar is roughly half that of the SMT processor, as shown in Table 3.3. Although misses in the superscalar hardware data structures are less frequent, because only one thread executes at a time, the superscalar lacks SMT's ability to hide latencies. As in all past studies of SMT on non-OS workloads [81, 43, 44, 45], SMTs latency tolerance more than compensates for the additional inter-thread conflicts in its memory subsystem and branch hardware. The lack of the superscalar's latency-hiding ability was most evident in the operating system, which managed to reach only 0.6 IPC in steady state! In contrast, user code achieved an IPC of 3.0. In addition, the superscalar squashed proportionally about twice as many instructions as SMT,

because the superscalar has only one source of instructions to fetch, i.e., the thread it is mispredicting.

In summary, including the operating system in superscalar simulations of a SPECInt workload perturbed bottomline performance more than on an SMT (a 15% vs. a 5% drop in IPC), because key hardware resources (the instruction cache and the L2 cache) were stressed several-fold and superscalar performance is more susceptible to instruction latencies. (In other hardware components performance drops were either smaller or reflected a large degradation to a previously well-behaved component.) This result suggests that researchers should be less confident of omitting effects of the operating system when evaluating superscalar architectures.

3.2.2 Evaluating Apache: An OS-intensive workload

Apache is the most widely deployed Web server. Its role is simple: to respond to client HTTP request packets, typically returning a requested HTML or other object. The objects are stored in a file-oriented database and are read from disk if not cached in the server's memory. We examine the Apache-based workload below.

3.2.2.1 The role of the operating system when executing Apache

Figure 3.5 shows the percentage of cycles spent in kernel and user mode for the Apache workload. This data differs significantly from the SPECInt multiprogramming workload in several ways. First, Apache experiences little start-up period; this is not surprising, since Apache's 'start-up' consists simply of receiving the first incoming requests and waking up the server threads. Second, once requests arrive, we see that Apache spends over 75% of its time in the OS, i.e., the *majority* of execution for Apache is in the operating system, not in application code.[1]

Figure 3.6 shows a high-level breakdown of the kernel cycles for Apache (shown as a percentage of total cycles), compared to the SPECInt start-up and steady-state periods. For Apache, the majority of its kernel time (57%) is spent executing system calls. That is, while the SPECInt workload is dominated by *implicit* OS use (responding to TLB-miss

1. [63] reported a similar ratio of user/kernel execution on a Pentium-based server.

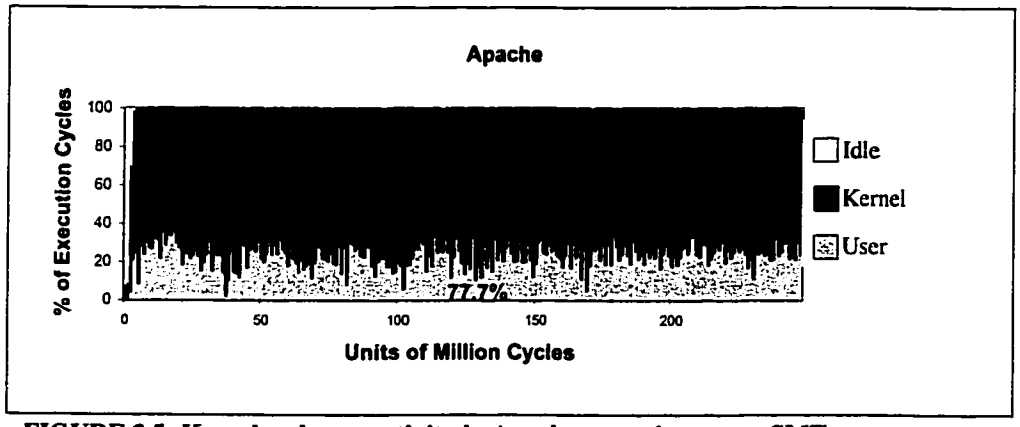


FIGURE 3.5: Kernel and user activity in Apache executing on an SMT.

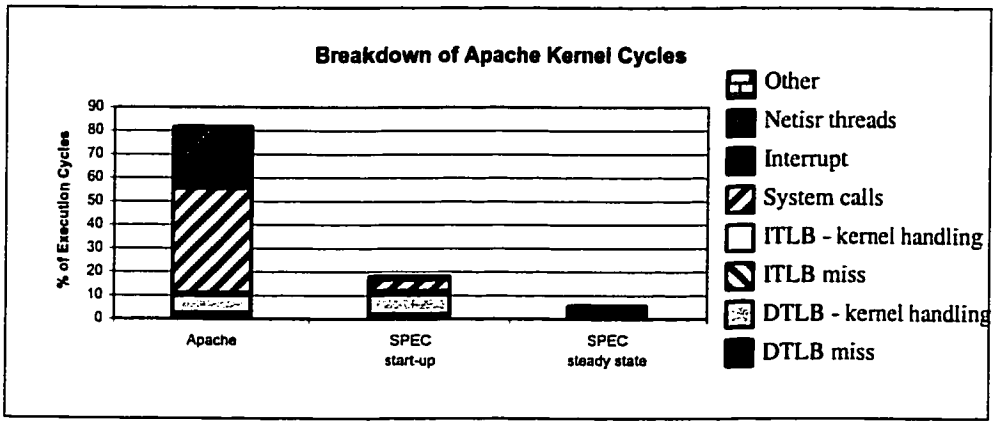


FIGURE 3.6: Breakdown of kernel activity in Apache on an SMT. Corresponding data from the startup and steady-state phases of the SPECInt workload are included for comparison.

exceptions), Apache uses the OS more *explicitly*. Apache also shows significant kernel activity that is initiated through network interrupts - there is no counterpart to this in the SPECInt workload. Apache spends 34% of kernel cycles (26% of all cycles) processing interrupt requests or responding to network interrupts in the *netisr* threads, the set of identical threads responsible for managing the network protocol stack on behalf of arriving messages. Only a moderate amount of kernel activity in Apache is due to DTLB misses (13%); in contrast, most of the SPECInt workload's kernel time is related to TLB miss handling (82% for steady state, and 58% for start-up).

Figure 3.7 shows a more detailed breakdown of the system calls for Apache. On the left-hand side, we see the percentage of all execution cycles due to each of the various system calls Apache executes. As the figure indicates, the majority of time is spent process-

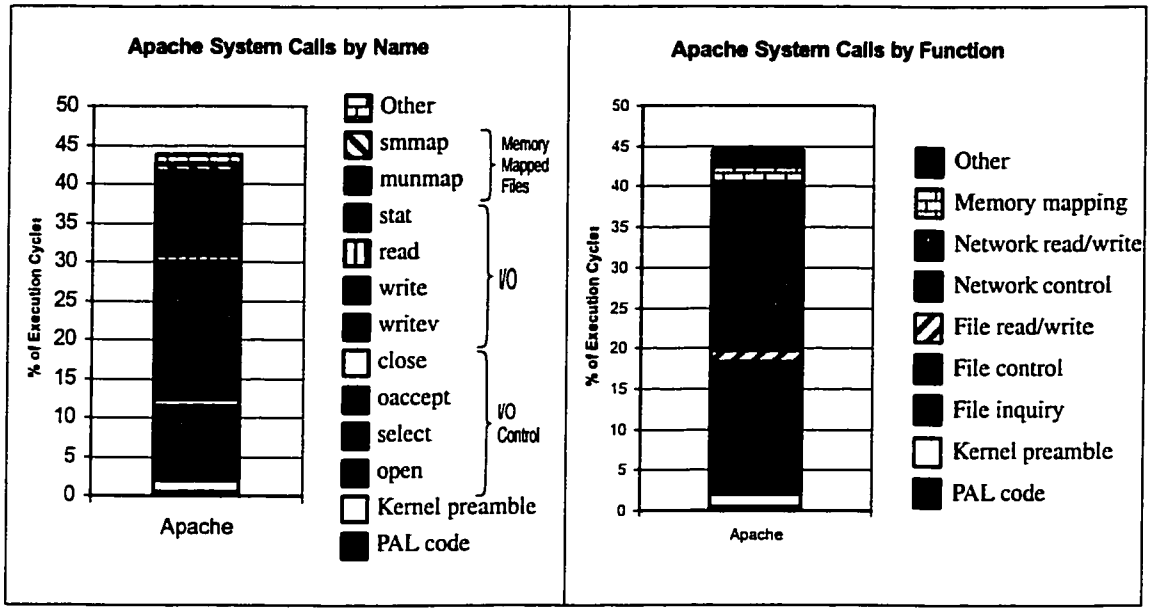


FIGURE 3.7: Breakdown of execution time spent processing kernel system calls on an eight-context SMT. The chart on the left shows the time spent in each system call. The chart on the right groups the system calls by activity. Note that the y-axis is the percentage of *all* execution cycles, not just kernel cycles.

ing calls to I/O routines: for example, Apache spends 10% of all cycles in the *stat* routine (querying file information), 19% of cycles in *read/write/writev*, and 10% of cycles in I/O control operations such as *open*. The right-hand side of Figure 3.7 shows a different breakdown of the same data. Here we qualify execution time by the type of resource - network or file - as well as the operation type. We see from this graph that network read/write is the largest time consumer, responsible for approximately 17% of all cycles and 22% of Apache's kernel cycles. As noted above, file inquiry (the *stat* routine) is the next largest consumer, followed by file control operations, which account for 6% of all cycles and 8% of kernel cycles. Overall, time spent in system calls for the network and file systems is nearly equivalent, with network services accounting for 21% of all kernel cycles and file services accounting for 18%.

3.2.2.2 Architectural performance characteristics

Table 3.4 shows a breakdown by instruction type for kernel and user code in Apache. In general, this is similar to the corresponding SPECInt table. The steady-state load/store percentages for Apache are closer to the start-up load/store percentages for SPECInt, because the SPECInt start-up includes a variety of OS services, while the steady-state

SPECInt workload is dominated by the TLB-handling routine. Overall, about half of all kernel memory access operations for Apache bypass the TLB, i.e., they specify physical addresses directly.

TABLE 3.4: Percentage of dynamic instructions when executing Apache by instruction type. The percentages in parenthesis for memory operations represent the proportion of loads and stores that are to physical addresses and do not use the DTLB. A percentage breakdown of branch instructions is also included. For conditional branches, the number in parenthesis represents the percentage of conditional branches that are taken.

Instruction Type	User	Kernel	Overall
Load	21.8	19.9 (54.2)	20.3 (42.0)
Store	10.1	11.5 (40.3)	11.2 (32.7)
Branch	16.7	17.8	17.6
Conditional	(54%) 70.6	(53%) 65.1	(52%) 66.2
Unconditional	12.9	16.0	15.4
Indirect Jump	16.3	13.7	14.2
PAL call/return	0.2	5.1	4.2
Total	100.0	100.0	100.0
Remaining Int.	51.4	50.8	50.9
Floating Point	0.0	0.0	0.0

Table 3.5 shows architectural performance characteristics for Apache and compares them to the SPECInt workload in steady state. The chart also shows statistics for Apache running on a superscalar. The Apache workload achieves an instruction throughput of 4.6 instructions per cycle on SMT (out of a maximum of 6), 18% less than the SPECInt workload. The causes of the lower performance are spread across most major hardware components, where Apache performs significantly worse than SPECInt. With the exception of the data TLB, all components of the memory subsystem experience more conflicts: e.g., Apache's L2 miss rate is 1.5 times worse than SPECInt's, its D-cache miss rate is 2.3 times worse, and its I-cache miss rate is 2.5 times worse.

The fetch unit also performs more poorly for Apache compared to SPECInt. On average, Apache has 20% fewer fetchable contexts than SPECInt, and sees many more instructions squashed. Apache also achieves 33% fewer cycles in which the six issue slots were fully utilized. However, despite these huge differences in memory and fetch system behavior, SMT still does a good job of tolerating latencies by handling more misses in parallel with the more demanding workload (last three rows).

TABLE 3.5: Architectural metrics comparing Apache executing on an SMT to SPECInt95 on SMT and Apache on a superscalar. All applications are executing with the operating system. The maximum issue for integer programs is 6 instructions on the 8-wide SMT, because there are only 6 integer units.

Metric	SMT Apache	SMT SPEC steady-state	Superscalar Apache
IPC	4.6	5.6	1.1
Instructions squashed (% of instructions fetched)	26.9	18.2	45.9
Avg. # of fetchable contexts	5.7	7.1	.4
Branch misprediction rate (%)	9.1	9.3	7.4
ITLB miss rate (%)	.8	.0	.7
DTLB miss rate (%)	0.6	0.6	0.2
L1 I-cache miss rate (%)	5.0	2.0	6.5
L1 D-cache miss rate (%)	8.4	3.6	3.4
L2 miss rate (%)	2.1	1.4	1.5
0-fetch cycles (%)	13.8	6.6	65.0
0-issue cycles (%)	3.1	0.6	62.4
Max. (6) issue cycles (%)	58.2	87.1	6.3
Avg. # of outstanding			
I-cache misses	1.9	0.9	0.5
D-cache misses	2.7	1.2	0.3
L2-cache misses	1.3	1.0	0.2

SMT's ability to hide latencies in Apache resulted in an average instruction throughput of 4.6 IPC - 4.2 times greater than the superscalar throughput, and the highest relative gain for any workload studied for SMT [43, 25]. The superscalar processor realized an IPC of only 1.1 - just 42% of the IPC it achieved for SPECInt. (In contrast, IPC for Apache on the SMT processor is 82% of what it realizes for SPECInt.) Most telling of the performance difference, the superscalar was unable to fetch or issue during more than 60% of the cycles, and it squashed 46% of the instructions fetched, due to branch mispredictions. SMT squashed fewer instructions, because multithreading reduces the distance that a mispredicted branch path will execute before the condition is resolved.

3.2.2.3 Inter-thread competition and cooperation

As mentioned previously, SMT can issue instructions from *multiple* kernel threads in a single cycle, which creates new potentials for inter-thread conflicts. Table 3.6 presents more detail on the miss behavior of Apache, focusing on the causes for conflicts. Com-

TABLE 3.6: The distribution of misses in several hardware data structures when simulating Apache and the operating system on an SMT. Bold entries signify kernel-induced interference: user-kernel conflicts are misses in which the user thread conflicted with some type of kernel activity (the kernel executing on behalf of this user thread, some other user thread or a kernel thread, or an interrupt).

	Percentage of Misses										
	Branch Target Buffer		**L1 Instruction Cache**		**L1 Data Cache**		**L2 Cache**		**Data TLB**		**Instruction TLB**
	User	**Kern**	**User**	**Kern**	**User**	**Kern**	**User**	**Kern**	**User**	**Kern**	**User**
Total miss rate	44.5	63.3	4.7	5.1	8.2	8.4	1.9	2.2	1.0	0.3	0.8
Miss cause	**Percentage of Misses Due to Conflicts (sums to 100%)**										
Intra-thread conflicts	12.2	**67.8**	6.4	**36.0**	5.8	**32.9**	.1	**13.9**	18.4	**7.3**	26.7
Inter-thread conflicts	.4	**13.8**	1.0	**28.6**	11.7	**32.4**	2.8	**27.1**	34.6	**7.3**	59.6
User-kernel conflicts	**2.4**	**3.4**	**13.6**	**11.6**	**5.0**	**5.0**	**13.0**	**9.3**	**8.6**	**13.7**	
Invalidation by OS			**.5**	**2.0**	**0**	**0**	**0**	**0**	**6.3**	**3.8**	**13.7**
Compulsory			.1	.2	.3	6.9	2.0	31.8			

pared to the SPECInt workload, most striking are the kernel/kernel and user/kernel conflicts, shown in bold. The highest cause of cache misses in Apache is conflicts within the kernel: 65% of L1 I-cache misses, 65% of L1 D-cache misses, and 41% of L2 cache misses are due to either intra-thread or inter-thread kernel conflicts. These misses are roughly evenly split between the two categories, except in the L2 cache, where kernel inter-thread misses are almost twice as numerous as intra-thread misses. User/kernel conflicts are very significant as well: 25% of L1 I-cache misses, 10% of L1 D-cache misses, and 22% of L2 cache misses are due to conflicts between kernel and user code or data.

The effect of running multiple kernel threads simultaneously on SMT can also be seen by comparing it with the superscalar, in which only one kernel thread can be active at a time. On a superscalar execution of Apache (data not shown), the percentage of misses due to kernel inter-thread conflicts are lower by 24%, 28%, and 38% for the I-cache, D-cache, and L2 cache, respectively, when compared to Apache on an SMT.

In the BTB, kernel intra-thread conflicts dominate, accounting for 68% of all BTB misses, while 6% of the misses are due to user/kernel conflicts. In contrast, it is user code that is responsible for the majority of misses in both TLBs (53% of data TLB misses and

86% of instruction TLB misses are due to user/user conflicts). This is despite the fact that user code accounts for only 22% of cycles executed.

While the data presented above concerns conflicts, executing threads simultaneously can result in constructive inter-thread behavior as well. In particular, prefetching occurs when one thread touches data that will soon be accessed by a second thread; the second thread will then find the data in the cache, avoiding a miss. It is interesting to compare the amount of such constructive sharing on SMT with the same behavior on a superscalar. Because there is finer-grained parallelism on SMT, there is more opportunity for this prefetching activity. Table 3.7 shows, for several resources, the percentage of misses

TABLE 3.7: Percentage of misses avoided due to inter-thread cooperation on Apache, shown by execution mode. The number in a table entry shows the percentage of overall misses for the given resource that threads executing in the mode indicated on the left-most column would have encountered, if not for prefetching by other threads executing in the mode shown at the top of the column.

Mode that would have missed	Misses avoided due to inter-thread prefetching as a percentage of total misses									
	Branch Target Buffer		L1 Instruction Cache		L1 Data Cache		L2 Cache		Data TLB	
	User	Kern	User	Kern	User	Kern	User	Kern	User	Kern
Apache - SMT										
User	0	0	8.7	0.2	1.7	0.1	7.7	0.4	0	0.1
Kernel	0	19.5	0.6	65.5	0.3	20.8	1.3	70.7	5.0	12.2
Apache - Superscalar										
User	0	0	3.4	0.5	2.5	0.5	4.8	1.1	0	0.04
Kernel	0	1.9	1.2	27.5	1.3	29.6	1.3	55.0	9.3	5.5

avoided due to constructive sharing in Apache. For example, on SMT, the overall miss rate of the L1 I-cache would have been 66% higher, had it not been for I-cache pre-loading of one kernel thread's instructions by other threads also executing in the kernel. In contrast, the effect of such sharing on a superscalar running Apache was only 28%. Again, the difference is due to SMT's executing multiple kernel threads simultaneously, or within a shorter period of time than occurs on a superscalar.

The impact of kernel-kernel prefetching is even stronger for the L2 cache, where an additional 71% of misses were avoided. Twelve percent of kernel TLB misses were avoided as well.

3.2.2.4 The effect of the operating system on hardware resources

Similar to the previous analysis of the SPECInt workload (Section 3.2.1.2 and Table 3.3), we now investigate the impact of the operating system on the cache and branch prediction hardware (Table 3.8[1]). The OS increased conflicts in all hardware structures, ranging from a 35% increase in the L1 data miss rate to over a five-fold increase in the L1 instruction miss rate. The increases roughly correspond to the conflict miss data of Table 3.6, i.e., the extent to which the user miss rate in a hardware structure degrades due to the additional kernel references is roughly proportional to the proportion of user misses caused by conflicts with the kernel.

TABLE 3.8: Impact of the operation system on specific hardware structures.

	SMT			Superscalar		
Metric	**Apache only**	**Apache+ OS**	**Change**	**Apache only**	**Apache+ OS**	**Change**
Branch misprediction rate (%)	4.4	9.1	2.1x	3.3	7.4	2.2x
BTB misprediction rate (%)	36.7	59.6	62%	31.1	55.3	77%
L1 I-cache miss rate (%)	0.9	5.0	5.5x	1.8	6.5	3.6x
L1 D-cache miss rate (%)	6.2	8.4	35%	2.9	3.4	17%
L2 miss rate (%)	0.6	2.1	3.5x	0.3	1.5	5x

With the exception of the superscalar instruction cache miss rate, the OS had a greater effect on the hardware structures when executing Apache than it did for the SPECInt workload. The difference occurs primarily because operating systems activities dominate Apache execution, but also because they are more varied and consequently exhibit less locality than those needed by SPECInt (the Apache workload exercises a variety of OS services, while SPECInt predominantly uses memory management).

3.2.3 Summary of results

In this section, we measured and analyzed the performance of an SMT processor, including its operating system, for the Apache Web server and multiprogrammed SPECInt workloads. Our results show that for SMT, omission of the operating system did not lead to a

1. Our simulators cannot execute Apache without operating system code. However, we were able to omit operating system references to the hardware components in Table 3.8, in order to capture user-only behavior.

serious misprediction of performance for SPECInt, although the effects were more significant for a superscalar executing the same workload. On the Apache workload, however, the operating system is responsible for the majority of instructions executed. Apache spends a significant amount of time responding to system service calls in the file system and kernel networking code. The result of the heavy execution of OS code is an increase of pressure on various low-level resources, including the caches and the BTB. Kernel threads also cause more conflicts in those resources, both with other kernel threads and with user threads; on the other hand, there is an inter-thread sharing effect as well. Apache presents a challenging workload to a processor, as indicated by its extremely low throughput (1.1 IPC) on the superscalar. SMT is able to hide much of Apache's latency, enabling it to realize a 4.2-fold improvement in throughput relative to the superscalar processor.

3.3 Related work

In this section, we discuss previous work in three categories: characterizing OS performance, Web server behavior, and the SMT architecture.

Several studies have investigated architectural aspects of operating system performance. Clark and Emer [17] used bus monitors to examine the TLB performance of the VAX-11/780; they provided the first data showing that OS code utilized the TLB less effectively than user code. In 1988, Agarwal, Hennessy, and Horowitz [1] modified the microcode of the VAX 8200 to trace both user and system references and to study alternative cache organizations.

Later studies were trace-based. Some researchers relied on intrusive instrumentation of the OS and user-level workloads [16, 48] to obtain traces; while such instrumentation can capture all memory references, it perturbs workload execution [16]. Other studies employed bus monitors [26], which have the drawback of capturing only memory activity reaching the bus. To overcome this, some have used a combination of instrumentation and bus monitors [78, 88, 79, 14]. As an example of more recent studies, Torrellas, Gupta, and Hennessy [78] measured L2 cache misses on an SMP of MIPS R3000 processors; they report sharing and invalidation misses and distinguish between user and kernel conflict misses. Maynard, Donnelly, and Olszewski [48] looked at a trace-driven simulation of an

IBM RISC system/6000 to investigate the performance of different cache configurations over a variety of commercial and scientific workloads. Their investigation focused on overall memory system performance and distinguishes between user and kernel misses. Gloy et al. [29] examined a suite of technical and system-intensive workloads in a trace-driven study of branch prediction. They found that even small amounts of kernel activity can have a large effect on branch prediction performance. Due to the limited coverage and accuracy of the measurement infrastructure, each of these studies has investigated OS performance with respect to only one hardware resource (e.g., the memory hierarchy). In contrast, we provide a detailed characterization of the interaction of user and kernel activity across all major hardware resources. We also quantify the effect of kernel activity on overall machine performance.

SimOS makes possible architectural studies that can accurately measure all OS activity. One of the first SimOS-based studies, by Rosenblum et al. [65], examined the performance of successive generations of superscalars and multiprocessors executing program development and database workloads. Their investigation focused on cache performance and overall memory performance of different portions of the operating system as different microarchitectural features were varied. Barroso, Gharachorloo, and Bugnion [9] investigated database and Altavista search engine workloads on an SMP, focusing on the memory-system performance. Other investigations using SimOS do not investigate OS activity at all [56, 85, 55, 34].

Web servers have been the subject of only limited study, due to their relatively recent emergence as a workload of interest. Hu, Nanda, and Yang [38] examined the Apache Web server on an IBM RS/6000 and an IBM SMP, using kernel instrumentation to profile kernel time. Although they execute on different hardware with a different OS, their results are roughly similar to those we report in Figure 3.7. Radhakrishnan and Rawson [63] characterized Apache running on Windows NT; their characterization is limited to summaries based on the hardware performance counters.

Previous studies have evaluated SMT under a variety of application-level workloads. Some workloads examined include SPEC (92 and 95) [82, 81], SPLASH-2 [44], MPEG-2 decompression [68] and a database workload [43]. Evaluations of other multithreading

and CMP architectures have similarly been limited to application code only [3, 35, 15, 2, 71, 41, 30] or PALcode [91].

Our study is the first to measure operating system behavior on a simultaneous multithreading architecture. SMT differs significantly from previous architectures with respect to operating system execution, because kernel instructions from multiple threads can execute simultaneously, along with user-mode instructions, all sharing a single set of low-level hardware resources. We measure both the architectural aspects of OS performance on SMT, and the positive and negative interactions between kernel and user code in the face of this low-level sharing. We also show how an operating-system intensive Web server workload benefits from simultaneous multithreading.

3.4 Chapter summary

In this chapter, we reported the first measurements of an operating system executing on a simultaneous multithreaded processor. For these measurements, we modified the Compaq/ DEC Unix 4.0d OS to execute on an SMT CPU, and executed the operating system and its applications by integrating an SMT instruction-level simulator into the Alpha SimOS environment. Our results showed that:

1. For the SPECInt95 workload, simulating the operating system does not affect overall performance significantly for SMT, although OS execution does have impact on a superscalar.
2. Apache spends most of its time in the OS kernel, executing file system and networking operations.
3. The Apache OS-intensive workload is very stressful to a processor, causing significant increases in cache miss rates compared to SPECInt.
4. From our detailed analysis of conflict misses, there is significant interference between kernel threads on an SMT, because SMT can execute instructions from multiple kernel threads simultaneously. On the other hand, there are opportunities for benefiting from cooperative sharing, as we showed in our analysis of inter-thread prefetching.

5. Overall, operating system code causes poor instruction throughput on a superscalar. This has a large impact for the Apache Web server, which achieves an IPC of only 1.1.
6. SMT's latency tolerance is able to compensate for many of the demands of operating system code. When executing Apache, SMT achieves a 4-fold improvement in throughput over the superscalar, the highest relative gain of any SMT workload to date.

Chapter 4

The cost of spinning on SMT

The previous chapter explored OS performance on SMT and introduced the modifications to a commercial OS necessary to allow it to execute on SMT. In this chapter, we pursue the goal of executing efficiently. We focus on the architectural interface that provides SMT-specific fast synchronization instructions, which can eliminate many forms of spinning.

Spinning occurs when one thread awaits an event (e.g., the release of a lock) by looping. Spinning occurs frequently in lock-based synchronization algorithms, but arises in other domains as well. These include the operating system idle loop and low-level device drivers, among others. A spin loop consists of a sequence of instructions, typically small, that loops, repeatedly testing if the event has occurred. On both a superscalar and multiprocessors, spinning can waste processor resources due to the opportunity cost of not context switching to another thread. Researchers have measured and developed techniques addressing the costs of spinning on these processors [37, 4, 40, 42, 5, 50, 10, 36, 90]; we do not investigate them here.

Spinning can exact a larger performance cost on SMT, because all threads share pipeline resources. We term pipeline resources as all resources shared between contexts that are necessary to execute instructions - that is, all shared resources except the caches and branch-prediction hardware. (As we show later, the spinning we examine has little impact on the caches and branch-prediction hardware.) Spinning threads impede other threads' progress by competing for pipeline resources. Worse, they can consume a disproportionately large fraction of pipeline resources, further delaying threads that are performing useful work. On SMT, spinning wastes context-private resources exactly as it does on a superscalar: it forgoes the possible benefit of context switching to another thread.

Spinning threads can dominate pipeline resources due to two compounding factors - software characteristics of spin loops and SMT hardware. First, as we shall demonstrate, the spin loops that we examine have high ILP and few long-latency instructions. Spin-loops enjoy a high cache and branch hit ratio, because they are typically small loops with localized memory access patterns. Second, at the hardware level, these software characteristics cause SMT's ICOUNT fetch policy [81] to fetch more often from and devote more execution resources to spinning contexts, because ICOUNT favours threads that make good progress.

SMT's hardware blocking locks [83] (SMT locks) provide a solution to spinning. The lockbox, described in [83], contains the logic to support hardware-level processing of locks. A thread that issues an SMT lock-acquire instruction will block, in hardware, until the lock becomes available. The processor fetches no instructions from the thread during this interval, and thus, the thread wastes no pipeline resources. Most spinning that awaits an event can be replaced with an SMT-lock-based routine in which the lock signals the event's occurrence. SMT-lock-based code should always outperform spin-lock-based code.

Although SMT locks potentially solve the problem of spinning on SMT, studying spinning is important for two reasons. First, replacing spinning with SMT-lock-based code requires recoding and recompiling. Understanding the cost of spinning enables developers to choose when the benefit of removing spinning outweighs the recoding cost. Second, this study of spinning provides insight into the more general question of how threads affect each others' performance at the pipeline level on SMT. A recent study examined thread interaction in the context of control speculation [77]; understanding thread interaction could help researchers determine how much work a low-priority thread can accomplish without affecting other threads' performance, for example.

This chapter investigates the impact of spinning on SMT and quantifies the performance benefit of replacing spinning with SMT-lock-based code. This study differs in focus from the original study introducing SMT locks ([83]). The previous study concentrated on improving synchronization latency to allow parallelization of five isolated loops. We concentrate on throughput of useful work, evaluate larger workloads, and explore in detail the interaction of spin and non-spin instructions.

As our workload, we focus on spinning in the operating system. As Chapter 3 observed, important SMT workloads such as web servers depend critically on operating system performance. We address two common examples of spinning in the OS: the simple lock-acquire routine[1] and the idle loop. We quantify their impact by comparing the performance of two versions of the operating system. The first version of the operating system was described in Chapter 2; it relies on spinning in the idle-loop and lock-acquire routines. We replace these sources of spinning with SMT-lock-based code to create the second OS version (we discuss OS modifications in Section 4.1.1).

Applications with both a high degree and a low degree of thread-level parallelism (TLP) can suffer from OS spinning. Applications with low TLP, including single-threaded applications, leave idle contexts that will spin in the idle loop. Applications with high TLP include server workloads such as the Apache web server. The high TLP and OS demands of server workloads can cause shared resource contention and thus lock-acquire spinning in the OS. Contention results because OS parallelism mirrors application-level parallelism, and threads executing in the OS share data extensively.

Our results show that spinning degrades performance by up to 75% on multiprogrammed workloads. However, surprisingly, spinning impedes performance much less than expected on a web-server-based workload. For example, on an Apache-like workload in which spinning constitutes 50% of instructions, removing spinning improved performance by only 13%. In general, we found that spinning only degrades performance to the extent that it occupies resources that non-spinning threads could have used effectively.

The remainder of this chapter proceeds as follows. Section 4.1 first describes additional simulation infrastructure and methodology used in the spinning experiments. Section 4.2 investigates spin loop behavior on a superscalar and on SMT. Section 4.3 explores the interaction of spin and non-spin code on an eight-context SMT. Section 4.4 examines the benefit of removing spinning on SMT. Section 4.6 covers related work, and we conclude in Section 4.7.

1. In addition to simple locks, there are read/write locks in the kernel. They cause a context switch rather than spin.

4.1 Methodology

This section describes methodology particular to the experiments in this chapter. Section 4.1.1 explains the OS and architectural modifications necessary to remove lock-acquire and idle spinning from the OS. Section 4.1.2 describes our evaluation workloads. Section 4.1.3 discusses the metrics with which we quantify spinning.

4.1.1 Eliminating spinning in the OS

Our experiments evaluate an OS that replaces two primary sources of spinning, the idle loop and the simple-lock-acquire routine, with SMT-lock-based equivalents. Removing spinning involved both software and architectural modifications.

At the architectural level, the SMT lock-acquire instruction, as originally specified in [83], lacks the functionality necessary to replace spin-based routines while preserving their semantics. It blocks a context from fetching instructions until the context successfully acquires the lock. However, the instruction's semantics must be extended to support hardware interrupts and mimic software time-outs to allow the OS to maintain control of the blocked context. Upon an interrupt, a blocked context must unblock and enter the interrupt handler, re-executing the lock-acquire instruction upon interrupt completion. The lock-acquire instruction must also support a time-out option to mirror the functionality of some spin loops that time out in software if lock acquisition does not succeed promptly. To provide this missing functionality, we modified SMT-lock semantics to read two dedicated registers upon execution. One holds a time-out value and the other holds a PC to which the processor branches if a time-out occurs.

With these two modifications, the SMT-lock instruction easily replaces the spin-based lock-acquire routine. The lock-acquire routine consists of a spin loop that attempts to acquire the lock and a retry counter (Section 4.2 discusses this spin loop in more detail). The retry counter lets the kernel time out and tend to other kernel functions if it fails to promptly acquire the lock. We replace this routine with two load instructions to prepare the time-out registers, followed by a single SMT lock-acquire instruction.

The idle loop consists of a more complex spin loop than the simple lock-acquire loop. It loops searching for work to do, indicated by eight different flags. Simultaneously check-

ing eight flags poses a problem for the SMT lock-acquire instruction, which can only monitor a single condition (i.e., if the lock is available). We chose an aggressive implementation that introduces a special, more powerful blocking instruction. This instruction blocks a thread until any of eight flags in memory are set. We eliminate spinning by simply inserting this instruction into the idle loop. A less aggressive implementation might avoid creating a new instruction at the expense of higher software overhead.

The idle loop also performs some low-priority, optimistic activities when no other work exists. These include pre-zeroing pages and testing memory. In general, SMT introduces a trade-off between performing these tasks and devoting the processor to threads doing useful work. However, these idle-thread activities did not occur in our experiments.

4.1.2 Base workloads

We explore each spin loop using a different workload, simulating all workloads 250 million cycles except where noted. The workloads are based on Apache and SPECInt95, both described in Section 2.1.4. We exercise the lock-acquire spin loop with a modified version of Apache, discussed in Section 4.1.2.1. We exercise the idle loop with a collection of SPECInt95-based multiprogrammed workloads, described in Section 4.1.2.2.

4.1.2.1 Lock-acquire spinning: Apache

Neither the SPECInt95 workloads nor Apache suffer naturally from excessive lock contention. The SPECInt95-based workloads perform little kernel locking because they predominately execute in user mode. In the Apache workload, spinning in the lock-acquire routines represents approximately 5% of all instructions. While investigating spinning's impact on Apache is important, we also seek to examine lock-acquire spinning in more spin-intensive workloads.

We create a set of workloads with a high degree of lock-acquire spinning by modifying the simulator to artificially induce spinning in Apache. Artificially induced spinning may differ from genuine spinning; we discuss some of the differences in the following paragraphs. However, it suffices for the purpose of examining spin- and non-spin-instruction interaction on SMT. Conveniently, it produces workloads that also vary only in the degree of spinning. This facilitates cross-workload comparisons.

We induce spinning by forcing the kernel lock-acquire routine to fail for a minimum number, X, of cycles. Following sections will refer to the number X as the 'lock-fail cycle constant.' The heart of the spin-based lock-acquire routine consists of a loop that checks the lock's status, waits until it becomes free and then attempts to acquire it. In the simulator, we force load instructions that check the lock's status to return a value reflecting 'lock in use' until X cycles elapse. In the SMT-lock-based version, we simply force the SMT lock-acquire instruction to block for X cycles. After X cycles elapse, both routines behave normally and acquire the lock, unless prevented from doing so by real contention.

This technique biases our results to slightly underestimate the performance improvement of removing spinning on SMT. To see why, consider two versions of a simple workload with two threads. Thread A executes in a critical section, and thread B waits to enter the critical section, blocking in one version and spinning in the other. The execution of the blocking version of the workload differs in two ways from that of the spinning version. First, thread A will complete the critical section sooner because it does not compete with thread B's spinning instructions for execution resources. Second, thread B will consequently wait fewer cycles before entering the critical section because thread A exits it earlier. Our methodology captures the first difference, but not the second. The lock-fail cycle constant, which corresponds to the number of cycles thread B waits to enter the critical section in this example, remains constant between the two versions of the workload rather than decreasing appropriately. We briefly explored a method to model this decrease, but it proved impractical.

4.1.2.2 Idle spinning: SPECInt95

We choose workloads of SPECInt95 applications that vary in the number of idle contexts. The idle time produced by these workloads does not suffer from the artificial artifacts discussed in the previous section. We omit Apache from idle-spinning experiments because a well-configured, fully-loaded web server should not idle. Table 4.1 lists the six different SPECInt95 workloads that we examine:

We selected Li and Compress for the single-application workloads because they exhibit the highest and lowest superscalar IPC of all SPECInt95 applications, respectively. They should bound the sensitivity of the remaining applications to spinning because they

TABLE 4.1: Multi-programmed workloads of SPECInt95 applications.

Workload Name	Number of Applications	Application Names	Superscalar IPC
Li	1	Li	3.6
Compress	1	Compress	1.5
Multi2	2	Li, Compress	
Multi4a	4	Li, Compress, Perl, M88ksim	
Multi4b	4	Li, Compress, Perl, Cc	
Multi7	7	Li, Compress, Perl, Cc, M88ksim, Go, Vortex	

utilize the processor the best and the worst of all the SPECInt95 applications. At the other extreme, we chose only one workload of seven applications because of the limited number of seven-application workloads possible (there are only eight SPECInt95 applications). We created the remaining workloads in the middle by mixing applications with a range of superscalar IPC between the two extremes. Idle periods exceeded 6000 cycles on average; consequently, idle-cycle measurements avoid the short-interval measurement problem discussed in Chapter 2.

4.1.3 Metrics to quantify spinning

While counting spin instructions is straightforward, measuring spin time defies a simple solution. Two factors impede meaningful time measurement. First, spinning can occur over short intervals. For example, a successful lock acquire consists of only eight instructions. No methodology we know of measures time accurately over short intervals, for reasons discussed in Chapter 2. The experiments here follow the technique described in that chapter and define time intervals as beginning and ending at the fetch stage of the pipeline. Second, multiple threads executing simultaneously on SMT complicate time measurement of any subset of executing threads due to inter-thread interactions.

We follow the methodology described in Section 2.2.2 and measure the percentage of total context-cycles to describe quantities such as percentage of cycles the processor spins. To compare the performance of spin-code versus non-spin-code in mixed workloads, we measure the weighted average of per-thread IPC. To measure the per-thread IPC of spin threads, for example, we first compute the per-thread IPC of each spinning thread by dividing the thread's spin instructions by the number of cycles over which it spins. We

then weight the per-thread IPCs of all spin threads by their contribution to the total number of context-cycles spent spinning to compute the average per-thread spin IPC. We refer to this quantity as simply per-thread spin IPC. When comparing spin and non-spin per-thread IPC, we will also include overall average per-thread IPC. (Overall per-thread IPC times the number of contexts equals total processor IPC.)

We compare per-thread IPC of spin and non-spin code to quantify the extent to which either disproportionately consumes processor resources. Comparing per-thread IPC differs in an important way from comparing relative instruction and cycle counts, although the two are related. The ratio of per-thread spin-loop IPC to the per-thread IPC of non-spin code relates directly to the fraction of overall spin instructions and spin cycles as follows:

$$\frac{IPC_{spin}}{IPC_{nonspin}} = \frac{\left(\frac{\text{spinInstructions}_{\%}}{\text{spinCycles}_{\%}}\right)}{\left(\frac{100-\text{spinInstructions}_{\%}}{100-\text{spinCycles}_{\%}}\right)}$$

Conversely, the ratio of spin cycles and instructions relates to per-thread IPC as:

$$\frac{\text{spinInstructions}_{\%}}{\text{spinCycles}_{\%}} = \frac{1}{\text{spinCycles}_{\text{frac}}\left(1-\frac{IPC_{nonspin}}{IPC_{spin}}\right)+\frac{IPC_{nonspin}}{IPC_{spin}}} \quad \text{(EQ 1)}$$

Where

$$\text{spinCycles}_{\text{frac}} = \frac{\text{spinCycles}}{\text{totalCycles}}$$

Equation 1 shows that comparing per-thread IPC tells a different story than comparing percentages of instructions and cycles. Comparing per-thread IPC characterizes threads' performance independent of the fraction of time over which the threads execute. The ratio of percentage of instructions to percentage of cycles, on the other hand, depends on the threads' fraction of total context-cycles. The independence of per-thread IPC more clearly captures how well the threads utilize the processor. For example, consider an eight-

context SMT in which the per-thread IPC of spin code and non-spin code equals 0.5 and 0.25, respectively. The ratio of per-thread IPC = 0.5/0.25 = 2. In other words, for every context-cycle that a context spins, it executes twice as many instructions as a context performing useful work. On the other hand, comparing the percentage of cycles and instructions depends on the number of spinning threads. Suppose that six contexts idle-spin and two contexts execute useful work. Using Equation 1, the ratio of percentage of instructions to percentage of cycles equals 1.14. If instead 2 contexts idle-spin and 6 perform useful work, that ratio increases to 1.6. Our results focus on per-thread IPC but also present numbers on percentage of instructions and cycles for comparison.

4.2 Pure spinning

We begin our analysis by first examining the characteristics of spin code in executions in which all threads spin. This analysis divorces the behavior of spin-code from interactions that occur between spin- and non-spin-code in mixed workloads.

We examine the behavior of three 100% spinning workloads on both a superscalar and an eight-context SMT: one in which the processor continuously spins in the lock-acquire loop, one in which it idles, and, for comparison, one in which it executes the base Apache workload. In the Apache workload, we factor out the effect of spin instructions from the hit-rate metrics. Pure spinning simulations executed for 15 million cycles or less because they require so little state. Table 4.2 compares the architectural performance of the three workloads.

TABLE 4.2: Architectural performance of spinning on a superscalar and 8-context SMT. We include the base Apache workload for comparison and factor out spin instructions from the hit-rate metrics for that workload.

Metric	Lock-acquire loop		Idle loop		Apache
	Superscalar	SMT	Superscalar	SMT	Superscalar
Overall IPC	4.0	4.0	4.6	5.0	1.1
Branch prediction rate (%)	99.9	100	99.9	100	92.6
I-cache hit rate (%)	99.9	100	99.9	100	93.5
D-cache hit rate (%)	99.9	100	99.9	81	96.7
IQ back-up cycles (%)	0.4	0.0	0.0	8.1	5.3
Average IQ size	7.6	6.0	8.5	26.3	7.0

On a superscalar, the two spin loops perform almost ideally in all hardware structures and achieve a high IPC of 4 and 4.6. Idle loop IPC slightly exceeds lock-acquire IPC due to better branch alignment. Between both routines, for every cycle spent spinning, the processor commits an average of 3.9 times more instructions than a cycle spent processing Apache and more than cycles spent executing any of the SPECInt95 applications examined in Chapter 3.

The lock-acquire spin loop achieves a high IPC of 4 because its locality allows it to capitalize on its high ILP. Figure 4.1 lists the main loop of the lock-acquire routine. The loop consists of only four instructions, of which the single load instruction repeatedly accesses the same memory location. High ILP results from two factors. First, there are few data dependencies, either within an iteration or across iterations. Within an iteration, testing of the lock status is independent of incrementing the retry counter. The only loop-carried dependence occurs between successive updates to the retry counter. Second, near perfect branch prediction accuracy eliminates control dependencies and lets the processor exploit parallelism across iterations. The lock-acquire loop cannot achieve an IPC higher than four because it fits in a single cache line, and a superscalar fetches at most one cache line per cycle.

```
lock_acquire:
        [ Test if lock is free and try to acquire. Branch to main_loop if lock is busy ]
main_loop:
        Load        a4,SL_DATA(a0)          # load lock status
        Subtract    t2,1,t2                 # decrease retry counter
        BranchBC    a4, lock_acquire        # branch if lock is free
        BranchGT    t2, main_loop           # loop back if counter hasn't expired

        # The retry counter has expired
```

FIGURE 4.1: Main spin loop of the lock-acquire routine. The retry count specifies the number of times to spin before jumping to a more complex spin loop that tends to other kernel functions.

On SMT, the idle-loop consumes slightly more execution resources than on a superscalar, but lock-acquire loop behavior remains the same. Idle-thread stacks align in the D-cache, increasing memory latency and consequently IQ backup cycles. The lock-acquire

threads all spin in the same I-cache line, preventing the single-ported I-cache from fetching from more than one thread each cycle. (We don't model hardware to multiplex a single I-cache line to multiple contexts in a single cycle). Fetch bandwidth limits lock-acquire loop performance, and thus the I-cache behavior reduces SMT performance to that of the superscalar.

4.3 Interaction of spin and non-spin code on SMT

This section examines the interaction of spin- and non-spin- instructions in the workloads described in Section 4.1.2. Section 4.4 will quantify how spinning degrades performance by eliminating spinning and measuring the performance improvement.

Figure 4.2 compares the per-thread IPC of spin and non-spin code on an eight-context SMT. Comparing IPC shows to what extent spinning contexts occupy a disproportionately large fraction of processor resources. The left half of Figure 4.2 covers lock-acquire spinning in Apache. In those graphs, the x-axis distinguishes workloads by the lock-fail cycle constant, i.e., the degree of induced spinning. The point labeled '0' corresponds to the base Apache workload with no induced spinning. The right half of Figure 4.2 covers idle-loop spinning in the multiprogrammed workloads. The number of idle contexts in the workloads range from one to seven. The bottom half of Figure 4.2 displays the percentage of spin cycles and instructions.

Both workloads vary over a wide range of spinning. Lock-acquire spinning accounts for only 5% of instructions in the base Apache workload, increasing to 50% of committed instructions as the lock-fail cycle constant increases (Figure 4.2c). The percentage of idle cycles reflects the number of idle contexts. For example, in the two left-most bars seven of eight contexts idle corresponding to the observed 87% of cycles that are idle (Figure 4.2d).

Lock-acquire spinning consumes a disproportionate fraction of processor resources. The per-thread IPC ratio of spin and non-spin code ranges from 1.8 with no induced spinning down to 1.4 as the amount of spinning increases (Figure 4.2a). Per-thread spin IPC progressively decreases partly because the spin loops become long enough to avoid the short interval measurement problem discussed in Section 2.2.2. This phenomenon may also explain the 2% increase in useful IPC in the graph. The high IPC of spin code may

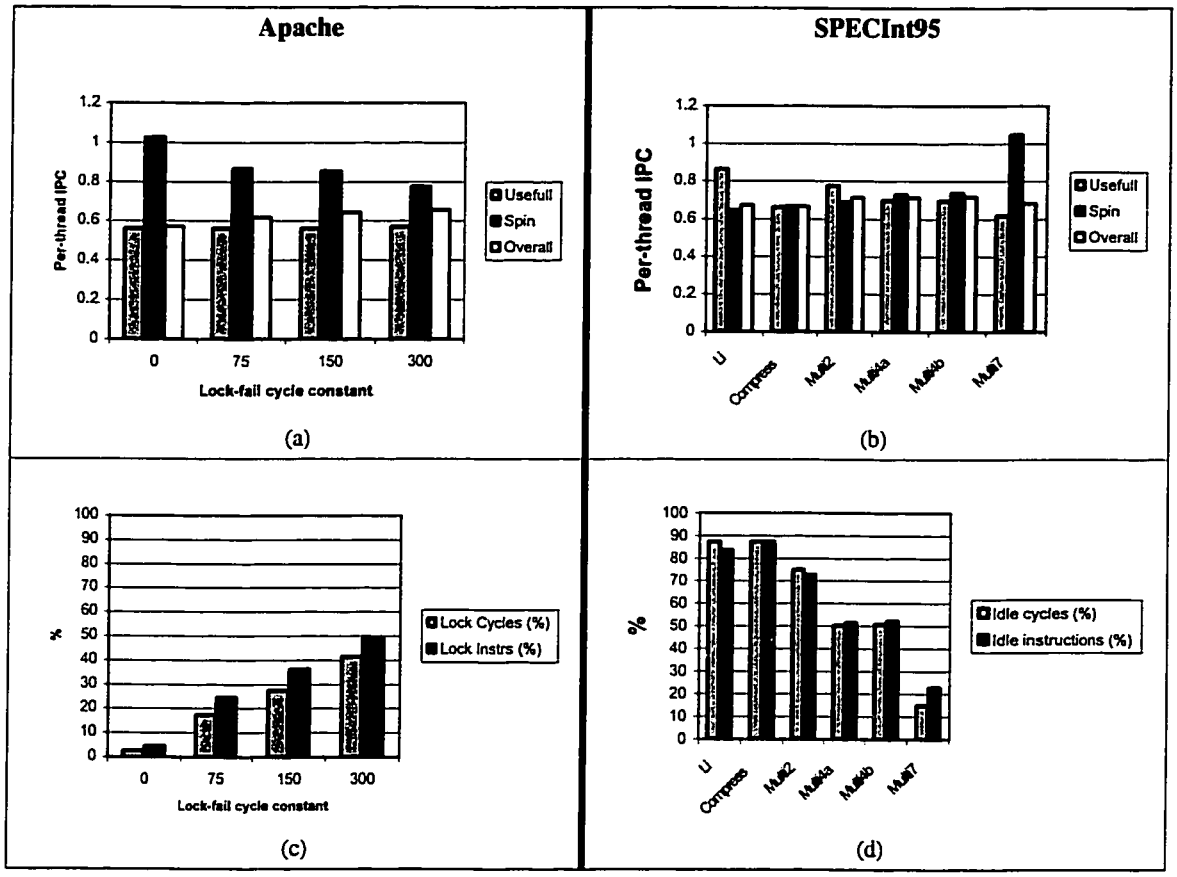


FIGURE 4.2: Characteristics of spin and non-spin code interacting on an 8-context SMT.

artificially lower the per-thread IPC of useful code. In other words, under-counting spin cycles (the bars on the left of Figure 4.2a) due to short spin intervals implies over-counting non-spin cycles.

In absolute terms, idle threads consume a large fraction of processor resources even when only one context idles (Figure 4.2d). The single idle thread in the Multi7 workload commits 23% of all instructions. It's instructions are low latency (more than 50% of cycles, less than four idle instructions reside in the IQ), causing it to be fetched from more often and thus commit more instructions than other threads. The percentage of idle instructions rises as the number of idle threads increases until, in workloads with seven idle contexts, idle instructions account for over 80% of all committed instructions.

Interestingly, idle threads consume an decreasingly disproportionate amount of processor resources as the number of idle threads increases. With only one idling context, idle per-thread IPC equals almost double non-idle per-thread IPC (1.05 versus 0.62). This ratio

decreases and reverses as the number of idle threads increases until, when seven contexts idle and one executes Li, idle per-thread IPC is only 0.7 times non-idle IPC (0.65 versus 0.85). The previous section showed that multiple idle threads conflict in the D-cache, limiting their IPC relative to useful threads as the number of idle threads increases. D-cache conflicts also explain why idle loop IPC exceeds Li's IPC on a superscalar while trailing Li's IPC on SMT.

4.4 Eliminating spinning on an eight-context SMT

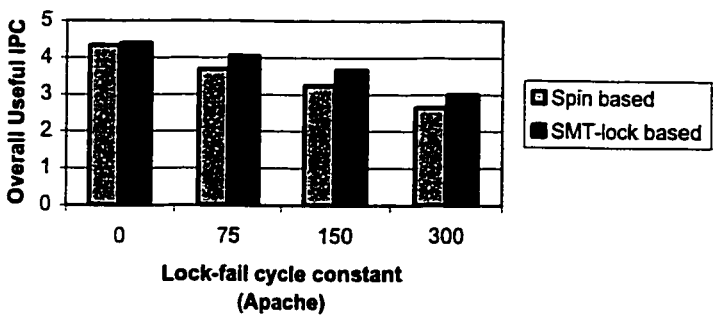
This section directly evaluates spinning's effect on SMT by replacing spin routines with their SMT-lock-based equivalents and measuring the performance improvement. We compare useful IPC (overall, not per-thread), which represents the rate of useful work per unit time. We first analyze lock-acquire and idle spinning on an eight-context SMT. Section 4.5 will examine lock-acquire spinning on a variety of other SMT configurations.

4.4.1 Lock-acquire spinning

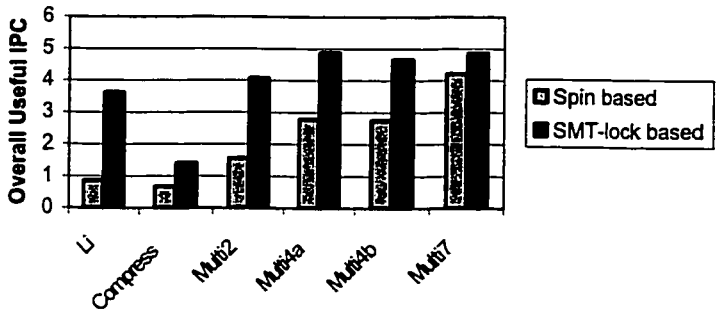
Figure 4.3 shows the impact of removing spinning from the Apache workloads in the top half and from the multiprogrammed workloads in the bottom half. The right-hand table summarizes the percentage increase in performance depicted in the graph.

Replacing spinning with SMT locks in the lock-acquire routine uniformly improves IPC. The improvement increases as the amount of spinning increases. Performance in the base workload (labeled '0') improves only slightly (1.4%) because spin instructions constitute only 5% of overall instructions. The benefit increases to 13.5% as the amount of spinning increases to 50% of committed instructions.

Removing spinning improves performance by so little because it uses resources that Apache would otherwise waste. In general, spinning harms performance only to the extent that it consumes resources that non-spinning threads could have used effectively. Chapter 3 demonstrated Apache's low superscalar performance. Although SMT improves throughput, Apache struggles on SMT for the same reasons as it does on a superscalar - high cache miss rate, poor branch predictions, etc. Apache's difficulty in effectively utiliz-



Lock-fail cycle constant	Improvement in useful IPC (%)
0	1.4
75	9.8
150	13.0
300	13.5



Workload	Improvement in useful IPC (%)
Li	321
Compress	110
Multi2	161
Multi4a	75
Multi4b	71
Multi7	16

FIGURE 4.3: Performance improvement of removing spinning on an eight-context SMT in both the Apache and SPECInt95 workloads. The tables details the percentage increase in overall useful IPC depicted in the graphs.

ing SMT leaves an opening for spinning threads to consume resources without interfering with Apache's performance as much as we might expect.

Table 4.3 provides more insight into spinning's limited impact on the workload in which spin instructions constitute 50% of committed instructions (labeled '300' in Figure 4.3). The table compares the pipeline performance of spin- and SMT-lock-based versions of the workload. We include corresponding data from the base Apache workload (labeled "0-cycle") for reference.

At the fetch stage, the percentage of cycles in which no instructions are fetched jumps almost 4x when spinning is removed, indicating that spinning instructions consume otherwise wasted fetch resources. Compounding waste in the fetch unit, Apache's poor branch prediction performance causes the percentage of wrong-path instructions to jump almost ten percentage points. Removing spinning effectively increases the branch misprediction penalty of non-spin threads. Spin instructions use few IQ entries: they constitute only 25%

TABLE 4.3: Pipeline performance.

Metric	300 cycle		0 cycle	
	SMT lock	spin	SMT lock	spin
0 instruction fetched cycles (%)	27.7	7.1	13.8	13.6
# of contexts fetchable/cyc	2.5	6.6	5.5	5.6
# of correct-path contexts fetchable/cyc	1.9	5.8	4.5	4.7
Useful (non-spin) instr squashed (%)	37.6	28.1	27.6	27.3
IQ backup cycles (%)	12.2	6.2	10.7	11.2
0 instructions retired cycles (%)	27.3	1.6	8	7.1

of instructions in the IQ on cycles in which it backs up. Without them, IQ back-up cycles double from 6% to 12% as high-latency non-spin instructions clog the IQ.

Two hardware structures that do not contribute to the explanation of spinning's limited impact are the cache and branch predictors. Their performance remains almost identical between the spin-base and SMT-lock-based version of the workloads.

4.4.2 Idle spinning

Idle spinning cripples performance of single-thread SPECInt95 workloads (bottom half of Figure 4.3). It costs a factor of four in performance for Li and a factor of two for Compress. Spinning hurts Li's performance more because Li has a higher superscalar IPC than Compress (3.6 versus 1.5) and so utilizes the machine more effectively. As the number of useful SPECInt95 threads increase, the amount of idle spinning decreases, and thus, its performance impact decreases.

The SPECInt95 workloads exhibit far more sensitivity to spinning than any Apache configuration. In the workloads labeled Multi4a and Multi4b, spinning constitutes roughly 50% of instructions. Removing spinning improves performance by an average of 73% compared to only 14% for the Apache workload with an equivalent amount of spinning. The greater sensitivity results from the SPECInt95 workloads' better ability to utilize the machine than Apache. The average useful IPC of the SMT-lock-based versions of Multi4a and Multi4b equals 4.5, compared to 3.0 for Apache.

Part of the idle loop's greater harm relative to the lock-acquire loop results from the idle loop's more disruptive nature. For example, in multi4b, conflicts with the idle loop caused one third of D-cache misses experienced by useful code. The miss-causing spin

instructions also impede performance by filling up the IQ and blocking the pipeline. On cycles in which the IQ overflows, idle instructions constitute 60% of IQ instructions. Removing spinning drops the D-cache miss rate experienced by useful code by one third, and cuts in half the percentage of cycles the IQ backs up.

4.5 Lock-acquire spinning on other SMT configurations

This section further explores the relationship between non-spin threads' ability to utilize the processor and spinning's impact by briefly evaluating the relative performance impact of lock-acquire spinning on four other SMT configurations. Table 4.4 lists the other configurations.

TABLE 4.4: Other hardware configurations evaluated.

Workload	Relative ability of Apache to utilize the processor
8-context SMT (perfect I-,D-cache, Branch)	More than on an 8-context SMT
12-context SMT	More than on an 8-context SMT
4-context SMT	Less than on an 8-context SMT
4-context SMT with fewer FUs	More than on a 4-context SMT

First, we evaluate Apache (lock-fail cycle constant equals 300) executing on an eight-context SMT with a perfect I-cache, D-cache and branch prediction hardware. Chapter 3 showed that Apache performs poorly in these structures; Apache should be able to more effectively utilize the processor when they perform ideally. Second, we evaluate Apache executing on a twelve-context SMT. The twelve-context SMT should translate the extra TLP compared to an eight-context SMT into better useful instruction throughput and hence greater sensitivity to spinning. Third, we consider a four-context SMT. Chapter 3 showed that Apache wastes hardware resources as the size of SMT, and thus the degree of TLP, decreases. This should translate into a decreased sensitivity to spinning. Finally, we consider a four-context SMT with only four integer functional units (versus the normal six). Apache should utilize that machine more effectively than a normal four-context SMT and consequently exhibit greater sensitivity to spinning.

As the number of contexts varies in these experiments, we alter the lock-fail cycle constant to maintain spin instructions as roughly 50% of committed instructions. A single constant would produce spinning that varies inversely with the number of contexts. For

example, with fewer contexts, the processor fetches from each context more often, resulting in a greater number of instructions committed per-context every cycle. The constant necessary to preserve the amount of spinning equals 420 cycles and 128 cycles on a twelve-context and four-context SMT, respectively. Table 4.5 lists results for the four workloads, including eight-context 300-cycle-constant results in the first column for comparison.

TABLE 4.5: Effect of lock-acquire spinning on other hardware configurations.

Metric	8-context SMT	8-context SMT (perfect I$, D$, branch)	12-context SMT	4-context SMT	4-context SMT (4 FUs)
Number of spin instructions (%)	49.2	43.4	46.8	47.4	42.3
Useful IPC (with spin locks)	2.6	3.3	2.9	2.2	2.0
(with SMT locks)	3.0	4.6	3.5	2.3	2.2
IPC increase by removing spinning (%)	13.5	37.4	19.0	6.9	12.1

On both the eight-context SMT with perfect hardware and the twelve-context SMT, Apache better utilizes the processor and consequently suffers more from spinning. The performance improvement due to removing spinning increases from 14% on a normal eight-context SMT to 37% on an eight-context SMT with perfect hardware and to 19% on a twelve-context SMT. In fact, these number are slightly conservative because the fraction of spinning instructions in both cases is slightly less than that on the eight-context SMT (43% and 47% versus 49%).

On the smaller, four-context SMT, spinning costs Apache less. Apache wastes more of the processor, as evidenced by the drop in useful IPC from 3.0 to 2.3 in the SMT-lock-based version of the workload. Removing spinning improved performance by only 7% versus 14% on the eight-context SMT. The performance impact increases to 12% on a four-context SMT with fewer functional units (4 FUs) compared to the normal four-context SMT. With fewer functional units, functional unit utilization by useful instructions increases from 40% to 57%. In other words, Apache can utilize a relatively larger fraction of the machine, making it more sensitive to spinning instructions.

4.6 Related work

Spinning's effect on multiprocessor systems has been the subject of many studies. Spinning degrades performance by competing for inter-processor shared resources such as the memory bus. Many techniques have been explored to reduce spinning's cost by reducing shared resource consumption; more recent investigations include [5, 50, 84, 36, 10]. For example, [50] proposed distributed locks. Distributed locks allow each processor to spin on a local flag rather than competing for a common flag. A processor releases a lock by clearing the flag of another processor, passing the lock to it. On a bus-based multiprocessor, no bus transactions occur while a processor holds the lock; only one occurs when a processor writes the flag of another processor.

In addition to increased bus contention, spinning also wastes resources due to the opportunity cost of not context switching to a thread that can perform useful work. A few papers such as [40, 42, 90, 89] investigate how to best choose when to context switch to another thread and how to schedule threads to reduce spinning cost. These techniques can work synergistically with the techniques to remove spinning evaluated in this chapter.

Several studies examine inter-thread interactions on multithreaded processors. [83] introduces SMT hardware blocking locks and compares its performance to traditional synchronization methods such as load-locked/store-conditional (a version of lock-free synchronization [36]). They concentrated on improving synchronization latency to allow parallelization of five isolated loops. We focus on the throughput of useful work, evaluate larger workloads, and explore in detail the interaction of spin and non-spin instructions. [70] explores improving job scheduling on SMT by incorporating information from hardware performance counters and sampling dynamic execution of application mixes to determine the best mix. They find that this kind of *symbiotic* scheduler can improve response time by up to 17% over schedulers that do not include symbiosis. [11] evaluated spinning's impact on a stylized fine-grained multithreaded processor, using a trace-driven simulation without caches. [81] compared various fetching policies on SMT and found that the ICOUNT mechanism produced the highest throughput, beating round-robin by 23%.

4.7 Chapter summary

This chapter examined spinning's cost on SMT under various workloads and processor configurations. Spinning has the potential to be especially harmful on SMT because all executing threads share pipeline resources. Spinning threads consume execution resources and slow down the progress of useful threads. SMT's ICOUNT fetch policy can exacerbate this effect, because it dedicates more pipeline resources to threads that make good progress and spinning threads make better progress than most other threads.

To quantify the impact of spinning, we replaced two common OS spin loops, the idle loop and simple lock-acquire spin loop, with non-spinning versions based on SMT hardware-blocking locks. We examined lock-acquire spinning with the Apache web server workload (modified to induce extra spinning) and idle spinning with a variety of multiprogrammed SPECInt95 workloads. Relative to the percentage of cycles, spinning threads consume a greater percentage of pipeline resources than either non-spinning Apache threads (user or kernel) or SPECInt95 applications.

Overall, we observed that spinning harms performance only to the extent that it consumes pipeline resources that non-spin threads could have used effectively. Chapter 3 showed that Apache has difficulty utilizing both superscalar and SMT processors. Executing on an eight-context SMT, removing spinning improved performance by only 1%, because lock contention causes only minor spinning and a loaded, well-configured web server idles little. However, even on a version of the Apache workload modified to induce lock-acquire contention so that spinning constituted 50% of committed instructions, replacing spinning with SMT-based lock-acquire routines improved performance by only 13%.

On different hardware configurations, the performance impact of spinning tracks the ability of Apache to utilize the machine. For example, relative to an eight-context SMT, spinning degrades Apache's performance less on a four-context SMT and more on a twelve-context SMT. This corresponds to Apache's higher throughput on the larger SMT and lower throughput on the smaller SMT.

SPECInt95 applications better utilize the processor and so exhibit more sensitivity to spinning. In workloads of four SPECInt95 applications executing on an eight-context

SMT, idle-loop spinning constituted 50% of committed instructions. Removing spinning improved performance by 75% compared to only 14% for the Apache workload with a similar amount of spinning. Single thread performance, an important SMT design consideration, suffers more from spinning. Removing spinning improved performance by between 2x and 4x for workloads in which a single SPECInt95 application executes on an eight-context SMT.

The relationship between the impact of spinning and the ability of non-spin code to utilize the processor applies to any code executing on SMT. For example, another thread could potentially accomplish work executing simultaneously with the Apache threads with little impact on Apache performance. Possible applications include deciding when to execute the optimizing routines in the idle loop (discussed in Section 4.1.1) or performance monitoring.

Chapter 5

Mini-threads: Increasing TLP on Small-Scale SMT Processors

This chapter explores the final software interface issue, that of architectural register usage on SMT. SMT increases TLP by sharing most processor hardware among executing threads. We extend this sharing to the register file, which serves to further increase TLP.

Recently, several manufacturers have announced small-scale SMTs (e.g., 2 to 4 thread contexts), both as single CPUs and as components of multiple CPUs on a chip [39, 75]. While these small-scale SMTs increase performance, they still leave modern wide-issue CPUs with underutilized resources, i.e., substantial performance potential is still untapped.

A primary obstacle to the construction of larger-scale SMTs is the register file. On the Alpha architecture, for example, an 8-context SMT would require 896 additional registers compared to a superscalar of similar structure. In terms of area, Burns and Guadiot [13] estimate that adding 8 SMT contexts to the R10000 would increase the register file and renaming hardware from 13% to 30% of the processor core. On Compaq's 4-context SMT, the Alpha 21464, the register file would have been *3 to 4 times* the size of the 64KB instruction cache [62]. In addition to the area requirements, the large register file either inflates cycle time or demands additional stages on today's aggressive pipelines; for example, the Alpha 21464 architecture would have required three cycles to access the register file [62]. The additional pipeline stages increase the branch misprediction penalty, increase the complexity of the forwarding logic, and compound pressure on the renaming registers (because instructions are in flight longer). Alternatively, lengthening the cycle time to avoid the extra pipeline stages directly degrades performance by reducing the rate at which instructions are processed.

This chapter proposes and provides an initial evaluation of a new mechanism to boost thread-level parallelism (and consequently throughput) on small-scale SMTs, without the commensurate increase in register file size. The mechanism, called *mini-threads*, alters the basic notion of a hardware context. On the hardware level, mini-threads add additional per-thread state (aside from general purpose registers) to each SMT hardware context. Using this hardware, an application can exploit more thread-level parallelism within a context, by creating multiple mini-threads that will *share* the context's architectural register set. We denote as $_{mt}$SMT an SMT with mini-threads, and use the notation $_{mt}SMT_{i,j}$ to indicate an $_{mt}$SMT that supports i hardware contexts with j mini-threads per context. For example, an $_{mt}SMT_{4,2}$ – a 4-context $_{mt}$SMT – has the potential to deliver the same thread-level parallelism as an 8-context SMT, but with half the number of registers and greatly reduced register-to-functional-unit interconnect.

Mini-threads improve on traditional SMT processors in three ways. First, mini-threads conserve registers, because each executing mini-thread does not require a full architectural register set. Second, $_{mt}$SMT allows *each* application the freedom to trade-off ILP for TLP within its hardware contexts. Applications can choose to use mini-threads to increase TLP, or to ignore them to maximize the performance of an individual thread. In the latter case, where the application dedicates its context to a single thread, the processor performs identically to SMT. Because of this, for single-program workloads, $_{mt}$SMT will always perform better than or equal to SMT.

Third, in addition to the savings in registers, mini-threads open up new possibilities for fine-grained thread programming. Each application can choose how to manage the architectural registers among the mini-threads that share them. For example, mini-threads can simply partition the architectural register set. A wide range of more complex schemes also exist, including sharing register values among mini-threads and even dynamically distributing registers to mini-threads as their execution-time needs change. While sharing the architectural register set among mini-threads within a context creates many opportunities to optimize architectural register usage, it also introduces some interesting problems and trade-offs as well, some of which we examine here.

The principal goals of this chapter are to (1) introduce the concept of mini-threads on SMT, mapping out the breadth of the design space it involves, and (2) perform an initial evaluation on one part of that design space to show that there is a potential performance gain from adopting $_{mt}$SMT. There are many possible ways in which mini-threads can be applied in both hardware and software. In this chapter, we chose to evaluate the most straightforward of these: static partitioning of the architectural register file among the mini-threads that share it. In particular, we examine *two-way* and *three-way* $_{mt}$SMTs, i.e., $_{mt}$SMTs with two and three mini-threads per context. Only if that scheme provides benefit is it worth exploring more complex schemes (and issues) that require a larger amount of effort, such as communicating and synchronizing through mini-thread-shared registers.

For all schemes, two opposing factors determine the performance of $_{mt}$SMT versus SMT. On the one hand, extra mini-threads per context may boost performance by increasing TLP, thereby increasing instruction throughput. On the other hand, performance may degrade due to additional spill code, since each mini-thread is limited to a subset of the architectural registers. $_{mt}$SMT wins when the TLP benefits of additional mini-threads outweigh the costs of fewer architectural registers per mini-thread.

This chapter evaluates these opposing factors in detail using five workloads: four applications from the SPLASH-2 parallel scientific benchmark suite and the multi-threaded Apache web server. These programs are naturally suited to $_{mt}$SMT because they explicitly control their degree of thread-level parallelism. In the bulk of the chapter, we quantify the factors that determine $_{mt}$SMT performance; for example, we provide a detailed analysis of the changes in spill-code (which can increase as well as decrease!) due to reducing the number of registers per mini-thread. Our results show a significant improvement for $_{mt}$SMT over SMT, averaging 43% on small SMTs of 4-contexts or less, and extending even to 8-context SMTs for some applications. Perhaps surprisingly, most applications suffer only minor per-thread performance degradation when two or even three mini-threads share a partitioned architectural register set. Thus, the increase in TLP due to the extra mini-thread translates directly into higher performance. This is particularly important for small-scale SMTs, which both need and enjoy the largest performance

gains. Small SMTs are also the most practical to build, positioning $_{mt}$SMT as an important technique for realistic implementations.

The rest of this chapter proceeds as follows. Section 5.1 defines the $_{mt}$SMT architecture, the mini-thread programming model, and their operating system and run-time support. In Section 5.2 we discuss the methodology of the simulator, workloads, and compilers that is particular to these experiments. Section 5.3 presents results that quantify the factors that contribute to $_{mt}$SMT performance: the benefits of adding thread-level parallelism and the costs of reducing the number of registers available to each mini-thread. In Section 5.4 we tie this analysis together with results that show the overall performance benefit of $_{mt}$SMT, when all factors are taken into account. Section 5.5 reviews previous research related to our study. We conclude in Section 5.6.

5.1 The $_{mt}$SMT architecture

$_{mt}$SMT improves on SMT by introducing architectural modifications and a thread model that optimizes architectural register usage. This section describes the basic $_{mt}$SMT architecture, its programming model, and operating system and run-time support for mini-threads.

5.1.1 $_{mt}$SMT architecture

An SMT processor can issue multiple instructions from multiple threads each cycle. An SMT *context* is the hardware that supports the execution of a single thread, such as a PC, a return stack, re-order and store buffers and exception handling and protection registers. Each context also maps a set of architectural registers, distinct from the architectural registers of all other contexts. A thread executing in one context cannot access the architectural registers of a thread in a different context.

$_{mt}$SMT augments SMT by adding to each context the hardware needed to support an additional executing thread *except for the registers*. The key feature of $_{mt}$SMT is that the mini-threads in a context *share* the context's architectural register set. Specifically, when two instructions from two different mini-threads in the same context reference the same

architectural register, they inherently reference the same *physical* register, accessing the same value. Note that the register renaming hardware has not changed – the mapping from architectural registers to physical registers proceeds exactly as it does on SMT. What has changed is the way that architectural register numbers are mapped to locations in the renaming table (and from there renamed to physical register numbers). Figure 5.1 depicts this register sharing on an $_{mt}SMT_{2,2}$. The mini-threads executing on the two PCs in each context map the same architectural register set.

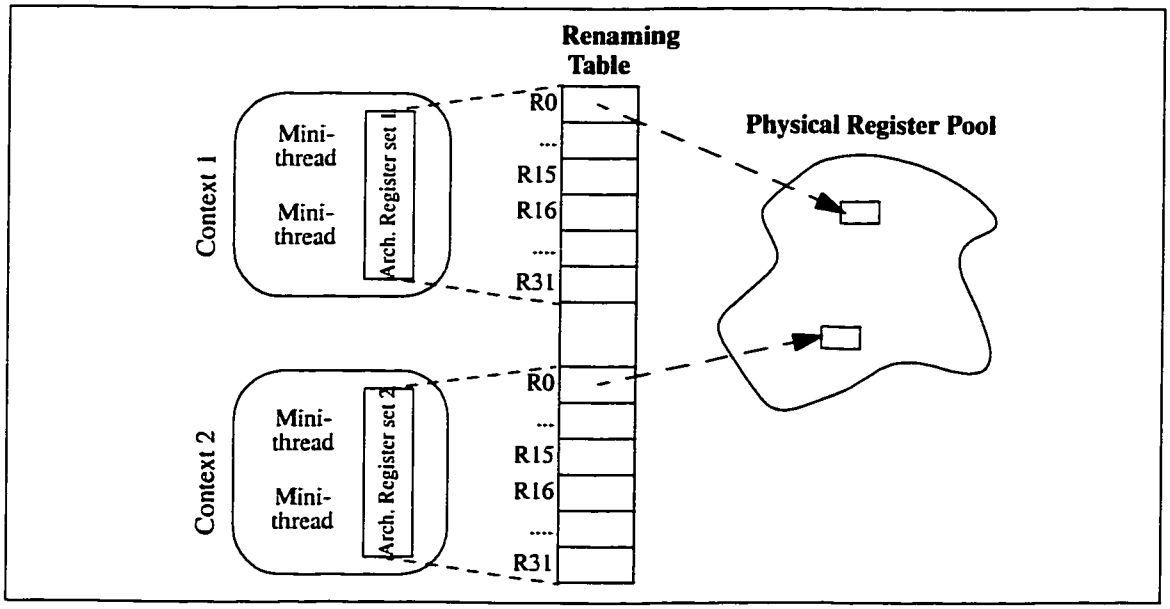


FIGURE 5.1: Register sharing among mini-threads on an $_{mt}SMT_{2,2}$. There are two hardware contexts, each supporting two mini-threads that share architectural registers within the context.

$_{mt}$SMT requires hardware similar to adding extra contexts to SMT. For example, a 4-context $_{mt}$SMT with 2 mini-threads per context closely resembles an 8-context SMT in terms of the number of mini-thread hardware resources, such as re-order and store buffers and return stacks. A few additional registers are also required beyond that on a 4-context SMT to support per-mini-thread exception handling and protection (~22 registers on the Alpha 21264 [19]). However, $_{mt}$SMT has the reduced register hardware, renaming complexity and register file access time of the 4-context SMT.

Although the implementation of architectural register sharing between mini-threads on $_{mt}$SMT requires minimal hardware modification, the sharing itself creates a very differ-

ent architectural interface. We begin with new terminology. Analogous to a context on SMT, a *mini-context* refers to the hardware necessary to execute a mini-thread. The architectural interface of a mini-context resembles that of an SMT context, including the architectural register set. However, on $_{mt}$SMT, all mini-contexts within the same context share this architectural register set. Therefore a mini-thread must manage the sharing of its registers in cooperation with the other mini-threads that execute within the same context. i.e., a mini-thread must be specifically compiled to execute in a mini-context.

The following section describes how mini-threads manage the sharing of their register sets.

5.1.2 $_{mt}$SMT programming model

On SMT, a program begins execution when the operating system first schedules it on a context, starting at the main program entry point. Later, the program may fork an additional thread by calling a *thread-fork* function and passing it the starting PC for the new thread. After the fork, the original thread continues executing in its context and the new thread begins execution on a different context. Each thread references its own distinct set of (architectural and physical) registers; threads communicate through shared memory.

On $_{mt}$SMT, a program begins in the same way, starting as a single mini-thread at the main program entry point and executing in one of the mini-contexts, using the *full* architectural register set. The program can ignore the additional mini-contexts, in which case it executes identically to an execution on SMT. Programs that choose not to use the additional mini-contexts will not incur any performance penalty relative to execution on SMT. The program is also free to fork a thread in a new (full) SMT context, as before.

Alternatively, a program executing on a mini-context can call a *mini-thread-fork* function. Instead of creating a new thread in a separate context, a *mini-thread-fork* creates a new mini-thread that shares the same context. The two mini-threads share the same architectural register set, and, because they were compiled as a mini-threaded program, they have arranged a priori (through the compiler) to coordinate their register usage.

Mini-threads can manage their registers in a variety of ways. Our models for register set sharing are similar to those proposed by Waldspurger and Weihl [86] for the April

coarse-grain multithreaded processor [2], which uses software context switching between threads. For example, two mini-threads may simply partition the register file in half, with one thread using the lower half of the register set, and the other using the upper half. (A larger number of mini-threads may similarly partition the register set more finely.) An alternative partitioned allocation would divide the register file unequally (one mini-thread receives fewer registers than the others). In a more complicated scheme, an application might reserve a few registers for shared values and partition the rest of the register set among the mini-threads. Sharing register values among threads can provide a low-latency communication and synchronization mechanism, or decrease the startup cost for new threads by reducing the register initialization step. Fast thread initialization and communication are necessary for the SMT optimizations researchers have suggested that require super-lightweight threads [92, 76, 18, 3, 72, 67]. Finally, all mini-threads in an application could share values in the entire architectural register set. $_{mt}$SMT permits all of these variations, because the application controls what register allocation is used and when. In all cases a compiler would have to compile a mini-thread for a specific scheme. $_{mt}$SMT also permits dynamically partitioning or sharing the registers should compiler or programming technology be developed to do this.

As an initial evaluation of mini-threads, this chapter focuses on one of these alternatives: statically partitioning each architectural register set among the mini-threads that share it. To explore the design space we consider both two-way and three-way $_{mt}$SMTs. Consider first an $_{mt}$SMT with two mini-threads per context. There are two methods to achieve this partitioning. In the first, each mini-thread is compiled for different architectural registers within a register set. An alternative strategy compiles both mini-threads for the *same* subset of the architectural registers and differentiates between them with an extra software-programmable state bit. The decode stage of the pipeline inserts this bit into the high-order bit of an instruction register field before accessing a register. Thus, the bit would be set to 1 on the mini-context using the upper half of the register set, and to 0 on the other mini-context. With this scheme, an application compiled to use the lower half of the register set will run correctly on either of the mini-contexts. Slightly more complex decode logic similarly allows an application compiled to use the lower third of the register

set (partitioning the register set in thirds leaves a few registers unused) to run correctly on all mini-contexts of a three-way $_{mt}$SMT.

5.1.3 Operating system and run-time support

The new architectural model introduced by mini-threads creates several new operating system and run-time support issues. First, because mini-threads execute within both run-time and OS procedures, those procedures must be compiled to use registers compatibly with the executing mini-thread. Second, when entered by one mini-thread, the OS must protect its registers from modification by other mini-threads executing in the same context, since they share the same architectural register set. Third, one or both of the OS and runtime must be modified to support mini-thread management operations.

We see two application environments for mini-threads and propose a run-time and OS support solution appropriate to each. One environment is dedicated and homogeneous, e.g., a dedicated server in which all threads run instances of the same code. The OS and runtime would be compiled specifically for this environment, to allow maximum concurrency. The second environment is heterogeneous, i.e., different contexts execute different programs, some multithreaded and some not. Here the standard OS and runtime could be used, with only one mini-thread in a context allowed to execute within them at a time (an approach also used on non-symmetric multiprocessors).

The first environment is exemplified by web servers. Here, all of the processes execute identical copies of the same server code, each handling a different request. The entire system is set up (statically) for that purpose (it is not, for example, running scientific programs at the same time). In this environment a single version of the OS and runtime executes, compiled to use half of the register set. The hardware partition bit described in Section 5.1.2 allows a single OS/runtime image to execute on either mini-context and isolates each mini-thread's registers from the other's. When the partition bit is used, the mini-contexts are indistinguishable from distinct contexts on an SMT, albeit with fewer architectural registers. Hence, OS and runtime code that manages contexts on SMT can manage mini-threads with minimal modification. (Most of the OS can be compiled automatically

to use fewer registers, except for the small number of assembly language files, which require manual register specification.)

This OS and runtime solution grants complete freedom to mini-threads to execute independently. Specifically, both mini-threads in a context may execute in the OS simultaneously, a performance-critical capability for OS-intensive workloads such as Apache (which spends 75% of its time within the OS). This performance advantage comes at the price of a loss in mini-thread flexibility to manage their common architectural registers. All mini-threads must partition the register set identically and cannot share any register values. Because of their homogeneity, mini-threads in web-server-like workloads perform well despite this restricted functionality.

The second environment we envision for mini-threads is more generic; for example, a multiprogrammed workload in which some programs use mini-threads and others do not. In this environment we execute a single operating system image compiled to use the *full* architectural register set, as on a standard SMT. The OS remains mostly ignorant of mini-threads, deferring mini-thread management to the runtime.

Applications that choose not to use mini-threads execute in the OS exactly as on SMT. However, when a mini-threaded application traps into the OS, the hardware blocks all other mini-threads in the same context. This guarantees that only one mini-thread per context executes in the kernel at a time, thereby protecting shared kernel registers from the actions of other mini-threads. On a normal SMT, the kernel saves the PC and registers on a trap before executing; for $_{mt}$SMT, we modify the trap handler to save the PCs, registers, and mini-thread IDs of both the *trapping* mini-thread and the *blocked* mini-threads within the context. After saving this state, the OS continues normal execution within the full register set; the mini-thread state is then restored on return to user mode. The OS remains ignorant of mini-threads, except for the extra state saved and restored on traps.

The Tru64 UNIX OS that we use already supports a form of scheduler activations [6], which allows the kernel to communicate thread activity to the pthreads runtime. A scheduler activation is a kernel upcall into the user-level threads package that alerts it about kernel scheduling events. This kernel-to-runtime communication is modified to also include information about blocked mini-threads in a context. If a trapping mini-thread blocks in

the kernel, the kernel communicates the PC, registers, and minicontext ID of the other (hardware-blocked) mini-threads as part of an upcall to the user-level runtime system. The pthreads runtime views mini-threads as simply user-level threads that have scheduling constraints, and hence, it requires few modifications. In our example, when the runtime receives the kernel upcall, it can place the hardware-blocked mini-threads on a queue; when the event that caused the trapping mini-thread to block in the kernel is resolved, the kernel again performs a runtime upcall, this time identifying the state of the trapping mini-thread to the runtime, which then schedules all mini-threads in the context to run.

In a statically partitioned, two-mini-thread environment this requires two versions of the runtime, one compiled for each register usage convention (i.e., 16 and 32 registers). Each application links to its appropriate version. There is ample precedence for multiple runtimes on a single platform. For example, Windows OS supports multiple versions of library routines; 32-bit and 64-bit programs executing on 64-bit x86-compatible processors have to link to different versions. Similarly, many applications already install private versions of library routines for their own use. In SPLASH-2-like workloads all threads are compiled to use the same runtime, and therefore avoid the issue of multiple runtime copies.

In summary, we take two approaches. For the server approach, which is OS-intensive, we recompile the OS and runtime to allow mini-threads in a context to execute them simultaneously, at the cost of limiting register-sharing flexibility among mini-threads. For the multiprogramming approach, we place no restrictions on register usage and sharing among mini-threads, but allow only one mini-thread within the OS at a time.

5.2 Simulation and evaluation infrastructure

In this section we describe the infrastructure and methodology used in our simulation-based experiments. We focus on elements specific to the experiments in this chapter; Chapter 2 described the general SMT simulation infrastructure, upon which the following is based.

5.2.1 $_{mt}$SMT simulation methodology

We emulate $_{mt}$SMT by compiling applications to use fewer registers and simulating the applications on a standard SMT. For example, to model an $_{mt}SMT_{4,2}$ we compile an application into threads that use only 1/2 the normal registers and simulate it on an 8-context SMT. This methodological simplification does not affect performance; each context touches no more registers than would be available on $_{mt}$SMT. We choose this methodology because it greatly simplifies compilation and allows us flexibility in choosing which specific registers to use. For example, on a 2-way $_{mt}$SMT, the mini-contexts that map the same register set cannot both use r30 as the stack pointer. Compiling for $_{mt}$SMT would require a compiler that is capable of using a different register for the stack pointer. By emulating $_{mt}$SMT on a larger SMT, we avoid having to modify the compiler to alter register semantics. Section 5.2.4 discusses compiler support in more detail.

5.2.2 Workloads

We evaluate $_{mt}$SMT on five programs described in Chapter 2: the Apache web server [7] and four applications from the SPLASH-2 benchmark suite [74].

Traditional throughput metrics such as IPC may not accurately reflect overall speedup for threaded programs. Instead, we use a higher-level performance metric of *work per unit time* as the basis for comparison. To define equivalent units of work, we modified each application to insert special markers at appropriate points in the code to indicate the progress that a particular thread has made. Our metric counts markers per unit time, where a marker corresponds to a unit of work. As the definition of work differs for each application, so does our method for inserting markers into the applications. For the Apache workload, total work corresponds to the number of requests processed, and the completion of each web request corresponds to a marker. For the SPLASH-2 applications, total work corresponds to application execution time. We create smaller units of work by instrumenting the inner loop of each of the SPLASH-2 applications to emit markers at fixed intervals throughout application execution.

5.2.3 Execution environment model

Modeling $_{mt}$SMT on SMT as described in Section 5.2.1 lets multiple mini-threads execute in the OS simultaneously. This corresponds to the OS-intensive environment described in Section 5.1.3. While this environment suits Apache, the SPLASH-2 applications would most likely execute in the environment in which other mini-threads in a context block when one traps. We do not explicitly model this blocking, but take it into account arithmetically. Our calculations suggest that the impact of blocking on our results would be less than 0.1%, because SPLASH-2 spends so little time in the kernel.

Briefly, the calculation proceeds as follows. We label a mini-thread as *free* when it is not blocked due to another mini-thread in the same context executing in the OS (we call this *buddy-blocking*). When free, mini-threads may be fetchable or may be blocked due to a different cause, such as an unavailable SMT lock. We compute the expected reduction in free mini-threads due to buddy-blocking and the effect of this reduction on performance (IPC).

Consider a two-way $_{mt}$SMT. We first derive P_{all}, the probability that, on any given cycle, both mini-threads in a context are in user mode (and hence are free). We define $frac_{OS}$ as the fraction of time that each mini-thread spends in the OS. Consider a average execution of length $(1 + frac_{OS})$. The amount of time that each mini-thread spends in the OS and at user level must obey the following three constraints:

1. On each cycle, either both mini-threads execute in user mode, or one mini-thread executes in the kernel and the other mini-thread is buddy-blocked.
2. For each mini-thread, the ratio of time spent in user-mode and kernel-mode (not including time spent buddy-blocked) must equal $frac_{OS}$.
3. For each mini-thread, the sum of time spent in user-mode, kernel-mode, or buddy-blocked must equal $(1 + frac_{OS})$.

These constraints determine that each mini-thread must spend $frac_{OS}$ time in the kernel and $(1 - frac_{OS})$ time executing in user-mode. Therefore:

$$P_{all} = \frac{(1 - frac_{OS})}{(1 + frac_{OS})}$$

The expected number of free mini-threads on a two-way $_{mt}$SMT equals $(1 + P_{all})n$, where n is the number of contexts on a 2-way $_{mt}$SMT. On an $_{mt}$SMT without buddy-blocking, the number of free mini-threads equals $2n$ (the number of mini-threads on the machine).

We convert the reduction in free mini-threads into an IPC impact by extrapolating from observed SPLASH-2 application sensitivity of IPC to the number of contexts on SMT. Consider two-way mini-threading on an 8-context SMT (an $_{mt}SMT_{8,2}$) - large SMTs spend the most time in the OS and so may be most susceptible to buddy-blocking-related performance degradation. On an $_{mt}SMT_{8,2}$, applications spend an average of 0.8% of cycles in the kernel, producing a $P_{all} = 0.99$. Consequently, the expected number of free mini-threads equals 15.9 versus 16 without the buddy-blocking restriction. We estimate the performance drop due to buddy-blocking by comparing the IPC performance of an 16-context SMT to a 15.9-context SMT. The performance drop averages only 0.08% over all the SPLASH-2 applications that we examine.

On a 3-way $_{mt}$SMT, the computation differs slightly but the estimated performance drop changes little. Repeating the above derivation for a 3-way $_{mt}$SMT produces:

$$P_{all} = \frac{(1 - frac_{OS})}{(1 + 2frac_{OS})}$$

The expected number of free mini-threads equals $(1 + 2P_{all})n$. On an $_{mt}SMT_{8,3}$, the SPLASH-2 applications spend an average of 1.2% of cycles in the kernel. This produces $P_{all} = 0.97$, and an expected number of free mini-threads equal to 23.5 (versus 24 with buddy-blocking). The estimated impact on IPC of this reduction equals 0.05%.

5.2.4 Compiler methodology

In this work we compile applications to use only 1/2 or 1/3 of the architectural register set for each $_{mt}$SMT configuration. We employed a combination of Gcc 2.95.3, a modern, widely-used, open-source compiler, and Compaq C 6.4, the closed-source compiler for Tru64 UNIX systems. Gcc provides a simple command-line option to control which architectural registers are available to the register allocator when compiling a program. We compiled the SPLASH-2 applications with Gcc.

Apache requires special compiler methodology. While the SPLASH-2 applications spend a negligible amount of time in the kernel (less than 1%), Apache spends a full 75% of execution cycles in the operating system. Because OS behavior dominates the performance of this workload, any evaluation of Apache must include it. Our SimOS-Alpha-based simulator executes Compaq Tru64 Unix 4.0d, a mature, commercial (shared-memory) multiprocessor-aware operating system. Incorporating this operating system into our evaluation required a methodology slightly different from that for the SPLASH-2 applications.

We compiled the OS for $_{mt}$SMT with Compaq's C compiler (Gcc is unable to compile it). The Compaq C compiler lacks Gcc's simple command-line option to control register allocation. Instead, it supports a set of pragmas that manipulate register usage, and require manual modifications to each OS source file. For this study, we modified enough files to cover 57% of all dynamic OS instructions when executing Apache. To compensate for the unmodified portions of the kernel, we extrapolated the trends observed over the modified regions of the OS to the entire OS, as described below. (Compaq C was unable to compile the OS to use 1/3 of the architectural register set, and so we do not report 3-way $_{mt}$SMT results for the Apache workload.)

We simplify the task of compiling for $_{mt}$SMT by carefully choosing which registers to mark as unavailable to the register allocator. We select registers from the pool of temporary registers, and retain the same proportion of callee- and caller- saved registers as in the original register convention. (Because we model $_{mt}$SMT on an SMT as discussed in Section 5.2.1, we are free to select registers in this way.) By selecting only temporary registers, we preserve the calling convention and other registers such as the SP, GP, and Ra.

Avoiding modification to the calling convention may lead to conservative performance results, because we miss the opportunity to optimize the calling convention for the smaller effective register set. However, there are portions of the workload, such as the shared libraries, that we did not recompile due to time limitations. Successful linking with this code required that we maintain the standard calling convention.

5.2.4.1 Shared Libraries

Twelve percent of Apache's dynamic instructions are from shared libraries. We factor out the effect of this code by extrapolating the behavior observed over the recompiled regions (e.g., non-shared library code) to the entire application. Many of our metrics examine trends as the number of available registers decreases. For instruction count metrics, we simply extrapolate changes in counts over the recompiled regions to the whole application. For example, if the dynamic instruction count of non-shared library code increases $x\%$ when the number of available registers decreases by half, we approximate that the dynamic count of shared library code would increase by the same percentage were we to recompile it. Extrapolating IPC requires more care, because it is a whole-processor metric (compared with instruction counts, which are maintained on a per-thread basis). At any cycle, there may be a combination of contexts executing recompiled code and shared library code. To estimate an IPC measurement for Apache compiled to use 1/2 the architectural register set, rather than attempt to define a meaningful IPC metric over recompiled code regions only, we perform the following operation. W.l.o.g., if the IPC of Apache on a 2-context SMT is x, the measured IPC of Apache on a ${}_{mt}SMT_{1,2}$ is y, and $z\%$ of Apache's dynamic instructions on ${}_{mt}SMT_{1,2}$ are from portions of code recompiled to use 1/2 the register set, then we estimate the IPC of Apache on ${}_{mt}SMT_{1,2}$ to be $x+\left(\frac{100(y-x)}{z}\right)$. This expression magnifies the IPC change $(y-x)$ by the relative amount of dynamic code that we did not recompile to use fewer registers (z). This approach assumes that as the number of available registers decreases, the impact on IPC of the unmodified code regions does not change significantly relative to the impact of the modified regions.

5.2.4.2 Measuring spill code

We measure the amount of spill code in applications by applying the following procedure. We first modified Gcc to emit a marker instruction before each spill instruction emitted by the register allocator. For the Splash applications, we then instrumented the binaries with Atom to count the markers, and ran the instrumented binaries as a single thread on an Alpha 21264 processor. For user-level Apache, we executed Apache on the superscalar simulator modified to count markers. We were unable to modify Compaq C to mark spill code and so could not directly measure kernel spill code by type. However, we report counts of overall stack loads and stores in the kernel, as we describe in Section 5.3.2.2.

There are a two types of instructions that might count as spill code, but that we do not include in our counts. First, due to instrumentation difficulties, we do not count instructions that allocate a stack frame for the sole purpose of storing spilled values (as opposed to storing local and temporary variables). Second, we do not count stack-based argument passing. However, ignoring argument passing does not affect the trends we measured, because the number of argument registers in the calling convention does not vary across our experiments.

Finally, note that the number of *non-spill* instructions can change as the number of architectural registers decreases. In the following sections, the term 'base compile' or 'base run' will refer to a compilation or execution of a program in which it can use the full architectural set of registers. Some instructions transform from non-spill code into spill code when registers are decreased. For example, in a base compilation, non-spill register shuffling may occur around procedure calls (even on a machine with infinite registers!). If, in a compilation with fewer registers, the values are shuffled to memory instead of other registers, the register moves become spill code.

5.3 Evaluating the register / mini-thread trade-off

$_{mt}$SMT allows applications to make a trade-off between one thread per context with a full architectural register set, or multiple mini-threads per context, each with a subset of the architectural register set. Two factors determine the efficacy of this trade-off: (1) the performance *benefit* due to the extra executing mini-threads, and (2) the performance *degra-*

dation due to fewer available registers per mini-thread. For each of these factors, there are also two levels of potential impact. At the hardware level, each can improve or degrade IPC, and at the software level, each can alter the number of instructions per unit of work. Of these four factors, two are straightforward. The extra mini-threads allow the processor to take advantage of greater TLP, possibly increasing throughput, while the reduction in the number of registers per mini-thread burdens the register allocator, possibly resulting in increased spill code. The other two factors reflect less obvious effects. First, the extra spill code can impact cache performance, affecting IPC. Second, an increase in the number of mini-threads can increase total thread overhead, altering the number of instructions executed.

These four factors completely describe $_{mt}$SMT performance relative to a traditional SMT processor. For example, Equation 1 expresses the speedup of a $_{mt}SMT_{2,3}$ over a base

$$\text{Speedup of } {}_{mt}SMT_{2,3} \text{ over base } SMT_2 = \left(\frac{IPC_{SMT_6}}{IPC_{SMT_2}}\right) \cdot \left(\frac{numInsns_{SMT_6}}{numInsns_{{}_{mt}SMT_{2,3}}}\right) \cdot \left(\frac{IPC_{{}_{mt}SMT_{2,3}}}{IPC_{SMT_6}}\right) \cdot \left(\frac{numInsns_{SMT_2}}{numInsns_{SMT_6}}\right) \quad \text{(EQ 2)}$$

2-context SMT, were *numInsns* denotes instructions per marker (our inserted units of work). In the equation:

- the first term corresponds to the IPC boost from extra TLP, due to extra PCs (contexts) on a conventional SMT,
- the second denotes the change in the number of instructions due to fewer registers per mini-thread (e.g., the cost of spill code),
- the third measures the IPC impact of that change on instruction count, and
- the fourth measures the extra instructions due to extra mini-threads (e.g., the cost of creating more parallelism).

The following two sections examine all four factors. Section 5.4 then presents overall $_{mt}$SMT performance, and quantifies the relative impact on performance of each of the factors discussed in Section 5.3.1 and Section 5.3.2.

5.3.1 Impact of the extra mini-threads

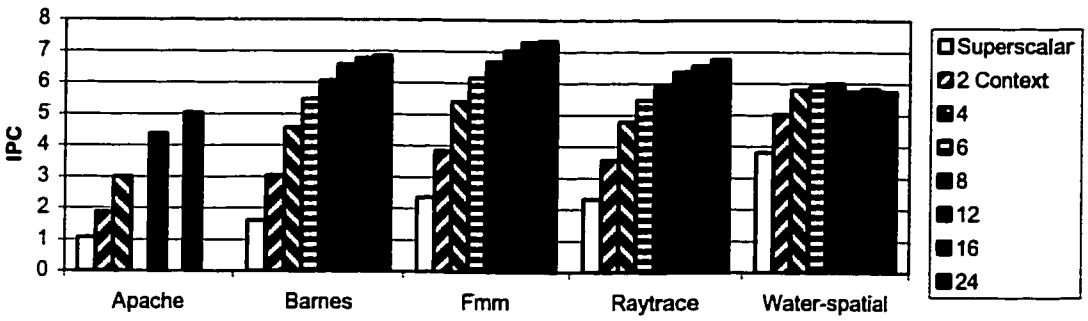
In this section we quantify the performance benefit due to the extra mini-threads. The extra mini-threads affect performance in two ways: the extra TLP can boost IPC and can alter the number of instructions per unit of work. We examine these two factors in the next two subsections, respectively.

5.3.1.1 Throughput improvement due to the extra mini-threads

This section measures the maximum throughput improvement due solely to the addition of extra mini-threads, i.e., due to the increase in thread-level parallelism. We can compute this effect by measuring the boost in throughput that is provided by a conventional SMT with the number of hardware contexts equal to the total number of mini-threads. For example, to calculate the maximum TLP benefit of $_{mt}SMT_{2,3}$ over the base 2-context SMT without mini-threads, we compare a 6-context SMT with a 2-context SMT. Comparing throughput on these two SMT machines isolates the throughput improvement from any effects of the reduced number of registers.

The top of Figure 5.2 graphs SMT throughput for all SMT sizes corresponding to the $_{mt}$SMT configurations that we evaluate, ranging from a superscalar up through a 24-mini-thread machine. The table at the bottom of Figure 5.2 shows, for each $_{mt}$SMT, the percentage improvement in IPC over its base SMT due to the extra mini-threads. Each entry in the table represents a loose upper bound on the potential performance improvement of $_{mt}$SMT. (The bound is not tight because suboptimal register allocation can also contribute slightly to the performance improvement, as discussed in Section 5.3.2.1.)

In most cases, doubling or tripling the number of threads (corresponding to a 2-way or 3-way $_{mt}$SMT, respectively) increases instruction throughput, and the benefit diminishes as the number of contexts increases. This is reflected by the levelling off of IPC in the graph and by the decreasing percentage IPC improvement with increasing base SMT size in the table. For example, on a 2-context SMT, the boost in IPC due to doubling or tripling the number of contexts averages 46% over all workloads (the maximum is 80%), while on an 8-context SMT, the benefit averages only 9%. In other words, extra contexts are most valuable for small SMTs, which have difficulty providing enough sources of



	$_{mt}SMT_{1,2}$	$_{mt}SMT_{2,2}$	$_{mt}SMT_{2,3}$	$_{mt}SMT_{4,2}$	$_{mt}SMT_{4,3}$	$_{mt}SMT_{8,2}$	$_{mt}SMT_{8,3}$
Apache	76	60		45		15	
Barnes	89	50	80	33	44	12	13
Fmm	62	41	60	23	30	9	10
Raytrace	53	34	54	24	33	10	14
Water-spatial	32	15	17	3	-1	-3	-4

FIGURE 5.2: Improvement in throughput due to extra contexts. The graph shows the IPC of a range of SMT sizes. The table shows the component of $_{mt}SMT$ performance solely due to the extra mini-threads. For example, the $SMT_{2,3}$ column shows the IPC improvement of a 6-context SMT over a 2-context SMT.

instructions to fully utilize the machine; but because the processor's execution resources are finite, the marginal benefit of additional contexts decreases as the number of contexts increases. Nonetheless, with the exception of water-spatial (discussed below), adding mini-contexts provides additional instruction throughput to any base SMT.

Apache and Water-spatial embody two extremes of sensitivity to the additional TLP provided by mini-threads. Apache benefits the most from extra contexts. Its poor superscalar performance deserves most of the blame; it achieves the lowest superscalar IPC of all the workloads, a plodding 1.1. The low superscalar IPC results from poor branch prediction, I-cache and D-cache performance. Due to these burdensome single-thread demands, the extra contexts propel throughput 4.7x, from 1.1 on a superscalar up to 5.0 on a 16-context SMT. (Chapter 3 discussed Apache's behavior on a superscalar and on SMT in detail.)

At the opposite extreme, Water-spatial squanders extra contexts. While increasing the number of contexts from one (a superscalar) to four boosts IPC 50%, performance languishes with any additional contexts. Even if reducing the number of available registers

were to cause zero degradation in performance, we see that extra contexts offer no further benefit. Part of the reason that Water-spatial does not successfully leverage increased TLP is its relatively high superscalar IPC. In general, the more a single thread can utilize a processor, the less opportunity exists for other instruction sources (i.e., other threads) to further increase utilization. Two other factors also limit Water-spatial's performance, and, in fact, cause IPC to *drop* as the number of contexts increases: the D-cache miss rate balloons from 0.3% on a 2-context SMT to 20% with 16 contexts, and the average percentage of cycles a context is blocked on a user-level lock rises from 17% to 25%.

The remaining three Splash applications benefit from additional contexts to a degree somewhere between Apache and Water-spatial. The average increase in IPC among Barnes, Fmm, and Raytrace ranges from 68% on $_{mt}SMT_{1,2}$ down to 11% on $_{mt}SMT_{8,2}$. Barnes benefits the most of the three, averaging a speedup of 46% over all $_{mt}SMT$ configurations, followed by Fmm at 34% and Raytrace at 32%. The degree to which each application benefits from additional contexts roughly tracks their superscalar performance for reasons discussed above.

5.3.1.2 Extra instructions due to extra mini-threads

In addition to increasing TLP, additional mini-threads can also affect performance by changing the number of instructions per unit of work. To measure this effect, we perform the same experiments as in Section 5.3.1.1, except that here we examine the change in the number of instructions (rather than IPC) as the number of contexts increases.

As the number of mini-threads increases, the number of instructions can both increase and decrease, and for different reasons. The increase is due to higher overheads associated with decomposing work and aggregating results among different threads. As the number of threads increases, the sum of fixed, per-thread management overhead across all threads increases. In addition, in the kernel component of the Apache workload, the number of instructions may increase because of increased inter-thread contention for OS resources. Sources of contention include the cache that maps file names to vnodes and the reader/writer locks that protect sockets. These complex locks do not spin; however, they incur software overhead under contention because of additional lock and thread management (they block the calling thread if the lock is unavailable). The number of instructions can

also decrease with additional mini-threads. For example, in work-pool-based parallel applications (such as Raytrace), a larger number of contexts could allow the application to more efficiently allocate units of work to the different threads. If this results in a more even division of processing among threads, then there will be fewer instructions spent raiding other threads' pools for more work.

Overall, the magnitude of this effect on instruction counts ranges from an increase of 6% on Fmm to a decrease of 6% on Raytrace. The effect is small, because the fundamental computation the applications perform is primarily not a function of the number of threads. Any changes in instruction counts will result only from effects of parallelization, which should be small if the application is already parallelized well.

5.3.2 Impact of reducing the number of registers

In the previous section we examined the performance contribution of the additional mini-threads. We now turn to the second side of the register/mini-thread trade-off, the effect on performance of the reduced number of architectural registers per mini-thread. As more mini-threads share an architectural register set, the number of registers available to each mini-thread decreases (as per our software convention of partitioning the register file). The reduced number of registers can affect performance in two ways. First, it can alter the number of dynamic instructions, for example, by increasing the amount of spill code. Second, this change in the instruction stream could in turn affect machine IPC, for example, by interacting with the caches.

Most compiler passes assume an infinite number of architectural registers. The register allocator is a notable exception. Because it maps a large number of pseudo-registers from the intermediate representation onto a much smaller set of architectural registers, it must insert additional instructions to move values both between registers and between registers and memory, so that all live values reside in registers at their use points. Spill code includes the traditional loads and stores to the stack, reloading global values to and from the heap, and register-to-register moves to transfer live values to the appropriate use registers. As the number of available architectural registers decreases, instructions of this nature should increase.

In addition to the register allocator, several optimizations depend on the number of architectural registers available. Some optimizations increase live ranges or the number of pseudo-variables and may actually degrade performance in register-constrained environments. In our compiler environment, the optimizations sensitive to the number of available registers are loop-invariant code motion, strength reduction, and common sub-expression elimination (CSE). Of these, Gcc disables only one CSE for simple constants (e.g., field offsets into data structures) in the register-constrained compilations in our experiments.

We begin in Section 5.3.2.1 by quantifying the number of extra instructions that result from the decrease in the number of architectural registers. In Section 5.3.2.3 we consider the effect of the spill instructions on IPC.

5.3.2.1 Extra instructions due to fewer registers

This section measures the third factor affecting the register/mini-thread trade-off: the change in the number of instructions per unit of work as the number of available registers decreases. To quantify the change, we compared each workload executing on two machines: an $_{mt}$SMT, and a conventional SMT with the same number of contexts as the $_{mt}$SMT has mini-contexts. The two machines differ only in the number of architectural registers available to each thread; comparing them therefore isolates the effect of reducing the number of registers per mini-thread.

Figure 5.3 graphs the percentage change in dynamic instructions due to fewer architectural registers for each $_{mt}$SMT configuration. For each application, seven bars arranged in two groups show the change in instructions for each configuration. The first four bars include all $_{mt}$SMTs that we evaluate in which the architectural register set is partitioned in half (two mini-threads per register set), and the last three bars include all $_{mt}$SMTs in which the architectural register is partitioned in thirds (three mini-threads per register set).

With the exception of Fmm, the applications are remarkably insensitive to the number of available registers. Compiling for a 2-way $_{mt}$SMT (i.e., with only 1/2 the architectural register set) results in an increase in dynamic instructions ranging between 16% for Fmm

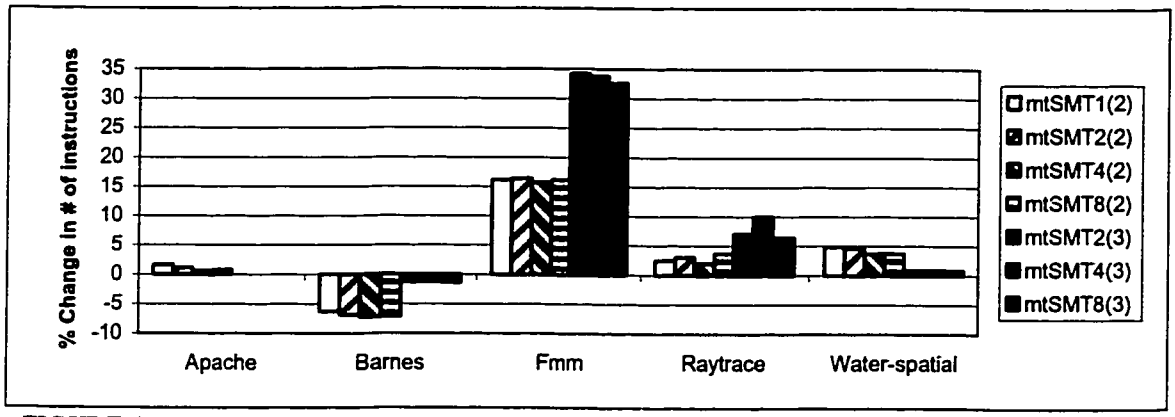


FIGURE 5.3: Change in instruction counts due to fewer registers per thread on $_{mt}$SMT. Each bar measures the percentage change in instruction counts between an $_{mt}$SMT configuration and an SMT which has the same number of contexts as the total number of mini-contexts in the $_{mt}$SMT.

to -7% for Barnes, with an average of 3%. On a 3-way $_{mt}$SMT, the increase ranges from 34% for Fmm to -3% for Barnes, averaging 11% across the SPLASH-2 applications.

For one application, Barnes, the amount of spill code *decreases* as registers become scarce – the instruction count in Barnes drops an average of 7% and 2% on the 2-way $_{mt}$SMTs and 3-way $_{mt}$SMTs, respectively. The entire reduction occurred in one procedure in which the register allocator substituted caller-saved registers for callee-saved registers when the number of architectural registers was reduced. Consequently, mandatory spills at procedure entry and exit were replaced with a smaller increase in spill code in the interior of the calling procedure.

The degree of mini-thread partitioning (2-way or 3-way) is the major factor that determines the amount of spill-code. Smaller variations exist within a partition. Because each workload is aware of the number of mini-threads, the execution profile of the workload can change as the number of mini-threads varies. This can alter the frequency of execution of both portions of code rich in spill code or code with little spill code.

The results for Apache provide our first glimpse into the sensitivity of the operating system to the number of available registers. In fact, the graph slightly overstates kernel sensitivity, because it combines user and kernel instruction counts, and Apache's user-level instruction count rise is larger (4%). Factoring out the user-level behavior leaves kernel instruction counts that barely budge upwards 0.8% as the number of architectural registers decrease! Two factors contribute to the kernel's insensitivity to a reduced register

set. First, the frequency of pointer usage in the kernel prevents the register allocator from keeping many values in registers. Second, simple operations with short-lived values, such as checking permissions or error conditions, dominate OS activity, leading to a low average number of simultaneously live values.

5.3.2.2 Breakdown of spill-code

This section expands the previous section's results by breaking down spill-code by type. We analyze the spill-code of application execution on a superscalar (primarily because we partially rely on Atom [20], which can not instrument multi-threaded programs). As discussed above, the execution profiles of applications vary with the number of threads, potentially causing superscalar spill-code measurements to vary slightly compared to the previous section's results.

Figure 5.4 breaks down spill code for all applications by instruction type as the number of available registers decreases. Bars labeled '32' indicate a base run in which the program executes with its full set of architectural registers (32 integer and 32 floating point). The bars labeled '16' and '11' (mini-threads share the register hardwired to 0, allowing three mini-threads to each use 11 registers in a 32 register set) measure the spill code in the application compiled for a 2-way and 3-way $_{mt}$SMT, respectively. We include the measurement of the application compiled with 24 registers (the '24' bar) to help illuminate trends. The stacked bars count the number of spill instructions by type, expressed as a percentage of total instructions of the base run. The upper line and the lower line express the total number of dynamic instructions and non-spill instructions on the scale at the right of the graph. We present Apache user and kernel data separately.

The Apache-kernel graph approximates spill code counts by counting all stack loads and stores. Since we compiled the kernel with Compaq C and did not have access to compiler sources, we could not modify it to mark spill code. Averaged over all applications compiled with Gcc (the four SPLASH-2 applications and user-level Apache), we found that loads and stores to the stack constitute 72% of spill code, and 75% of stack loads and stores are spills. Thus, counting all stack loads and stores roughly captures the amount of spill code.

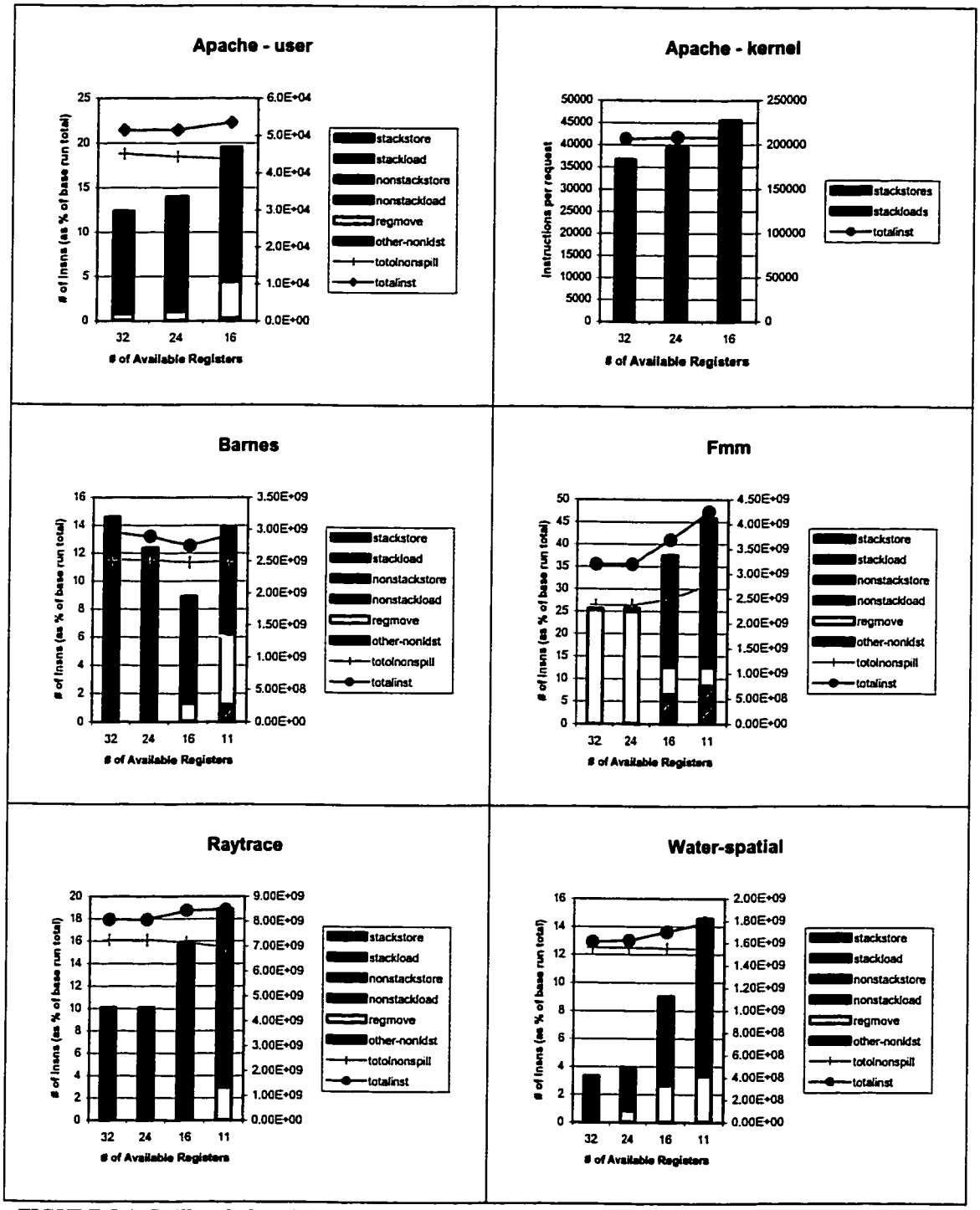


FIGURE 5.4: Spill code breakdown on a superscalar. The stacked bars count the number of spill instructions by type, expressed as a percentage of total instructions of a base run (with 32 registers). The number of available registers decreases along the x-axis. The right-hand y-axis counts total dynamic instructions for all applications except Apache - there it counts instructions per web request. The upper and lower lines count total instructions and non-spill instructions only, respectively. The count of stack loads and stores for Apache-kernel is over all instructions, not just spill code.

Over most applications, spill code constitutes between a modest 10% and 20% of total dynamic instructions. There are two exceptions. First, in Fmm spill code begins at a high 26% of instructions in the base run and increases to 35% as available registers decrease. (The percentage numbers in the graph differ slightly because they are with respect to the number of dynamic instructions in the base compile, and the number of non-spill instructions increases.). Second, water-spatial has the fewest relative number of spill instructions - only 3% are spills in the base run.

As mentioned above, for most programs, loads and stores to the stack constitute the bulk of spill instructions that are generated with the 32-register compile. Within some executions, the number of stack-load spills and stack-store spills differ. Sources of extra stack-stores include spilling a newly defined value to memory for the first time. Extra stack-loads may occur when an unmodified value is read from the stack at different points in the code (e.g. reloading read-only variables). Some applications also contain a small number of spills that load from non-stack locations. Reloading of global symbols is one common source of these loads. Overall, as the number of available registers decreases to 1/2 and 1/3 of the register set, the rise in stack operations causes the total number of loads and stores to increase slightly from an average of 32% to 37% and 38% of all instructions, respectively.

As the number of available registers decreases, increases in non-load-store operations contribute the most to rises in spill code. For example, in user-level Apache, the rise in the number of non-load-store spills with 16 registers accounts for 62% of the total increase in user spill code. Two effects contribute to the increase in non-load-store spill code as the number of available registers decreases. First and foremost, the compiler generates more register-to-register moves to shuffle values within the restricted set of architectural registers. Second, the register allocator chooses to undo simple CSE optimizations and recompute some constant values rather than spill them to memory, thereby generating extra non-load-store operations. Fmm is an exception: with 32 registers, almost all spill code consists of non-load-store instructions, shifting to stack loads and stores with fewer registers.

Figure 5.5 presents the same spill-code data as Figure 5.4, but distinguishes between spills around procedure calls by both the caller and callee, and other spills. For all programs except Fmm, procedure call handling constitutes the bulk of spill instructions in the

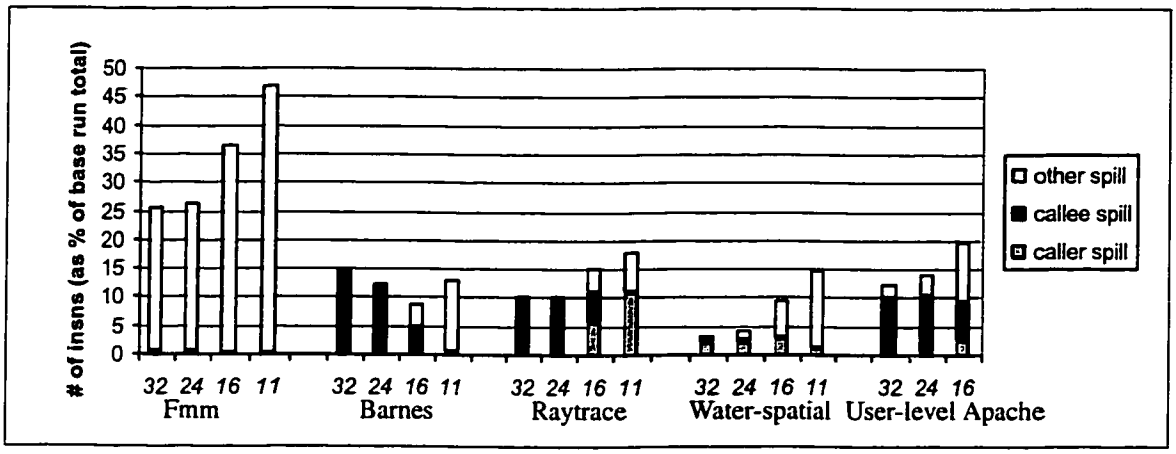


FIGURE 5.5: Breakdown of spill code around procedures. This figure breaks down the spill code measurements depicted in Figure 5.4 into spilling around procedure calls, both by the caller or callee, and other spills.

base execution. As the number of available registers decreases, spilling within a procedure rather than around procedure calls begins to dominate. Fmm is an exception: 99% of its spill code occurs within procedure bodies, resulting from basic conflicts in live ranges.

The high percentage of callee-saved register spilling across applications reflects an implementation decision in Gcc. Although Gcc's register allocator allocates caller-saved registers before callee-saved registers, it always allocates all callee-saved registers before it will spill any values[1]. The register allocator MUST save callee-saved registers at the beginning and end of procedure calls, while it only saves caller-saved registers over their live range. In some cases, refraining from using a callee-saved register and storing a value in a caller-saved register may reduce overall spilling, even if no caller-saved registers are available and some other value must be spilled to free a register.

5.3.2.3 Effect of spill code on IPC

In this section we quantify the final factor that contributes to $_{mt}$SMT performance: the impact on IPC that results from decreasing the number of available registers. To determine the magnitude of this factor, we perform the same experiments as in the previous sub-section, this time examining IPC instead of dynamic instruction counts. For each $_{mt}$SMT con-

1. Thanks to Joern Rennecke from the gcc mailing list for pointing out that: "The register allocator makes no attempt to use fewer callee-saved registers than are available."

figuration we compare the IPC of $_{mt}$SMT with the IPC of an SMT machine with a number of contexts equal to the total number of mini-threads on the $_{mt}$SMT. The two differ only in that the $_{mt}$SMT has fewer available registers per mini-thread. Figure 5.6 graphs the percentage change in IPC due to the reduction in registers.

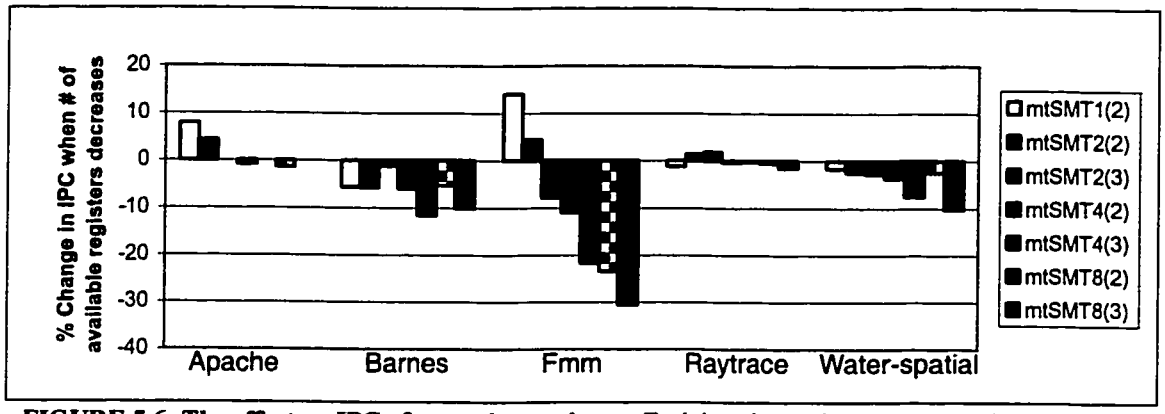


FIGURE 5.6: The effect on IPC of removing registers. Each bar shows the percentage change in IPC between an $_{mt}$SMT and a normal SMT with the same number of contexts as the $_{mt}$SMT has mini-contexts. The only difference between the two machines is the number of available registers per thread.

For most programs and most $_{mt}$SMT configurations, reducing the number of available registers causes a reduction in IPC. The extra instructions compete for machine resources with the non-spill instructions. This competition lowers IPC primarily by increasing conflicts in the L1 data cache and the DTLB. For example, reducing the number of registers to support mini-threads increases Fmm's D-cache miss rate by 2.5x (on average), causing the observed 31% drop in IPC on $_{mt}SMT_{8,3}$. The 12% IPC dip for Barnes on $_{mt}SMT_{4,3}$ reflects a 7 percentage point rise in D-cache misses. The smaller drops in IPC for the other applications similarly reflect lower degradations in D-cache performance. A higher DTLB miss rate can also lower IPC, because mini-contexts stall waiting for the miss-causing instructions to reach the top of the active list after a TLB miss trap.

Some application/$_{mt}$SMT configuration combinations bucked the general trend. For example, the IPC of FMM on an $_{mt}SMT_{1,2}$ jumped 14%, due to fewer backups in the floating point instruction queue. When restricted to fewer registers, the compiler replaced register shuffling to the floating point registers with integer operations such as spilling to the stack, which improved the balance between floating point and integer instructions.

5.3.3 Summary

Overall, adding mini-contexts increases TLP, significantly boosting IPC in most, but not all configurations. We observe increases in IPC solely due to this effect ranging from 89% down to -4%. On the other hand, reducing the number of available registers per mini-thread generally degrades performance due to increased spill code. The extra spill code both increases the dynamic instruction count and negatively impacts IPC by raising the number of D-cache and DTLB misses. The magnitude of each of these spill-code effects on performance ranged from a 31% decrease to a surprising 14% improvement. In the next section we compose these results into an overall picture of $_{mt}$SMT performance.

5.4 Performance on $_{mt}$SMT

The previous sections examined in isolation the four factors that contribute to $_{mt}$SMT performance: the IPC and instruction count impact of both (1) extra TLP from mini-threads and (2) a reduced number of registers per mini-thread. This section addresses two remaining questions. First, how do the four factors relate in importance to each other? And second, does $_{mt}$SMT improve overall performance?

To enable an intuitive comparison of the different factors, in this section we present a stacked bar chart (Figure 5.7) that combines the impact of each factor on $_{mt}$SMT's performance relative to SMT. We cannot graph the factors directly, because they are related multiplicatively in Equation 1, and a stacked bar chart expresses additive relationships. To convert the relationship of the factors to an additive one, we take the logarithm of Equation 1, representing each log term as a bar segment. The advantage of graphing the logarithm of each factor is that the relationship between bar heights becomes intuitive. Two factors of equal magnitude will be the same height, and if they have different signs, they will cancel each other's effect on performance.

The triangles in Figure 5.7 signify the overall performance improvement of $_{mt}$SMT over SMT, taking into account the factors' positive and negative effects. The magnitude of a bar segment measures the contribution to speedup of a particular factor. Note that the height of any bar or combination of bars can only be interpreted against the y-axis if it is transposed to the origin (because the height of each bar represents the logarithm of a term

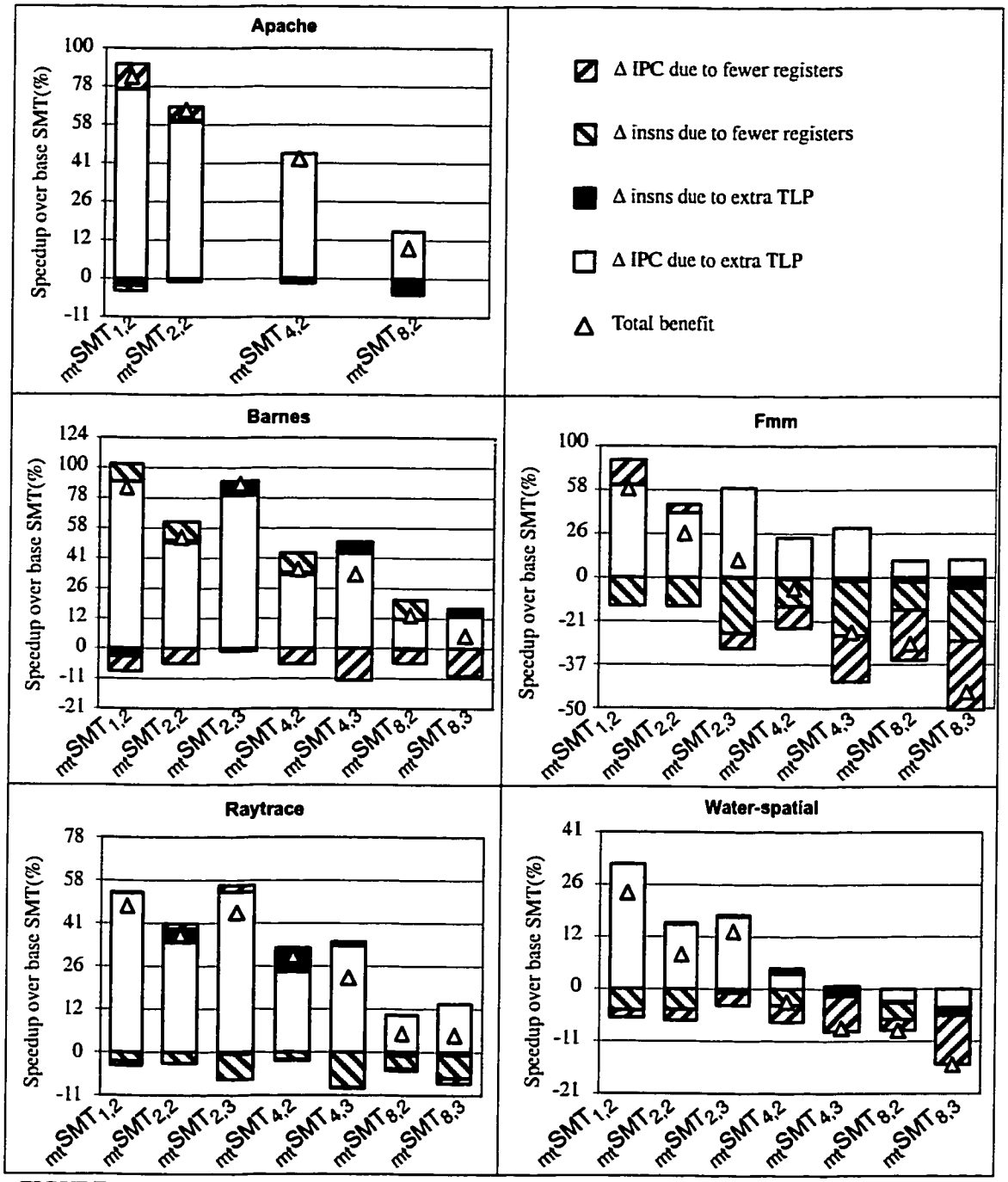


FIGURE 5.7: Performance improvement of $_{mt}$SMT over SMT, broken down by factor. The x-axis enumerates the $_{mt}$SMT configurations, arranged in order of increasing total number of mini-contexts. Each column consists of four bars and measures the speedup of $_{mt}$SMT over the corresponding base SMT. The four segments of each bar measure the change in performance due to each factor. 'Δ IPC due to fewer registers' and 'Δ insns due to fewer registers' denote changes in IPC and dynamic instruction count due to fewer available registers per mini-context. 'Δ IPC due to extra TLP' and 'Δ insns due to extra TLP' denote changes in IPC and dynamic instruction count due to the increase in number of mini-contexts. The y-axis indicates percentage speedup. Note that the scale differs for each application. The triangle in each column represents the sum of the heights of the bars which equals total speedup.

in Equation 1). Table 5.1 echoes the triangles in the graph, reporting the total percentage $_{mt}$SMT speedup.

Apache benefits the most from mini-threads and these benefits appear on all $_{mt}$SMT

TABLE 5.1: Total percentage $_{mt}$SMT speedup.

	$_{mt}SMT_{1,2}$	$_{mt}SMT_{2,2}$	$_{mt}SMT_{2,3}$	$_{mt}SMT_{4,2}$	$_{mt}SMT_{4,3}$	$_{mt}SMT_{8,2}$	$_{mt}SMT_{8,3}$
Apache	83	66		43		10	
Barnes	85	53	88	36	33	14	5
Fmm	60	26	10	-6	-25	-30	-45
Raytrace	48	37	45	29	22	5	5
Water-spatial	24	8	14	-3	-8	-9	-15

configurations. However, the greatest performance improvement occurs on the smallest SMTs, where TLP is the most constrained. For example, adding an extra mini-thread to a superscalar (the bar at the left of the graph) increases Apache's request throughput by 83%. As the size of the SMT grows, the performance improvement due to mini-threads progressively decreases. However, even on an 8-context SMT, trading off registers for contexts boosts request throughput by 10%[1].

The SPLASH-2 applications also benefit from mini-threads, although on average less than Apache. Mini-threads improve the performance of all applications on the smaller SMTs. For example, speedups on a 2-context SMT lie between 14% and 87%, with a median improvement of 36%. For half of the applications, the improvement also scales to larger SMTs. Barnes and Raytrace see speedups averaging 33% with 4 contexts and 8% with 8 contexts. The other two applications do not benefit on either of the larger SMTs, with average performance degrading by 5% and 20% on the 4- and 8-context SMTs, respectively.

The magnitude of each factor explains why performance improves so much. For most applications and most $_{mt}$SMT configurations, the IPC boost due to extra mini-threads far dominates any other factor. With the exception of Fmm and Water-spatial executing on larger SMTs, the IPC benefit of the extra mini-threads averages 47%. Reducing the num-

1. At 16 contexts, hardware context 0 becomes a performance bottleneck because certain OS activities such as network interrupts are funneled through it, resulting in 20% idle time on other contexts.

ber of available registers has little effect on performance, allowing the IPC boost to translate directly into improved overall $_{mt}$SMT performance. The performance decline from decreasing by 1/2 the number of registers available to each mini-thread (effect on IPC combined with executing more instructions) averaged -2%. The impact of restricting each mini-thread to a third of the register set averaged -9%.

For Fmm and Water-spatial, the large cost of reducing the number of registers and the relatively small boost in IPC due to extra mini-threads meant that $_{mt}$SMT did not pay off for larger SMT configurations. The overall impact of reducing the number of registers averaged -37% for Fmm and -8% for Water-spatial. In contrast, the overall benefit of the extra mini-threads averaged only 15% and -1%, respectively, insufficient to overcome the cost of the reduced number of registers.

Surprisingly, the IPC impact of the increase in instructions equaled or exceeded that of the instructions themselves. In 40% of all configurations and applications examined, the IPC impact of fewer registers exceeded the percentage increase in the dynamic instruction count. In 70% of all configurations, the IPC impact was at least 50% of that of the instruction count.

Augmenting SMT with greater than two mini-threads per context improved performance only on small SMTs. On a two-context $_{mt}$SMT, three mini-threads raised the average performance improvement compared to SMT to 43% from 31% with two mini-threads. However, on larger SMTs, they performed worse than two mini-thread $_{mt}$SMTs, because larger SMTs benefit less from extra mini-threads and the even further reduced number of registers induced more spill code.

Overall, our results show that trading off registers for contexts improves performance, especially on the smaller SMTs that are characteristic of the first commercial implementations. In particular, performance improved by an average of 48% (averaged over all applications) on a 2-context SMT, with decreasing improvements on successively larger SMTs. In the above experiments, we forced applications to use mini-threads. If we allow them instead to use mini-threads only when advantageous (as they can do, since employing mini-threads is an application-specific decision), then the average performance improvement on 4- and 8-context SMTs is 22% and 6%, rather than 20% and -2%, respectively.

5.5 Related work

Many researchers have explored methods for conserving registers in various contexts. Closest to our work is Waldspurger and Weihl's study of register relocation [86] in the April/Alewife processor, a distributed shared-memory multiprocessor that uses software multithreading to tolerate latencies from remote memory references and failed synchronization attempts [2]. April executes one thread at a time until it incurs a miss, at which point a hardware trap signals the OS to context switch to another thread. Waldspurger and Weihl propose treating the four hardware contexts of the April CPU as a single large register file, partitioning the entire file in software. Their scheme uses a *register relocation mask* mechanism to offset thread-local register numbers into the large register file. The compiler must ensure protection between all of the threads executing on the machine; hence, all loaded threads are assumed to be part of a single application. Waldspurger and Weihl evaluate the trade-off between the context-switch cost that their software multithreading necessitates and processor utilization using a synthetic workload with stochastic run lengths and varying inter-fault latencies.

In contrast, we focus on the impact of the number of available registers on spill code versus processor utilization, and we evaluate this in a more realistic and modern hardware and software environment. We evaluate intra-context architectural register partitioning among mini-threads for an out-of-order, simultaneously-multithreaded CPU with multiple hardware contexts and register renaming. We simulate real parallel programs and a multithreaded server compiled for mini-threads, include operating system code, and we provide detailed measurement and analysis of all of the factors that influence performance.

Several researchers have investigated adding special-purpose light-weight contexts with reduced register requirements to a superscalar, mostly for the purpose of improving performance of a primary thread by prefetching or warming up the branch prediction hardware [31, 8, 92, 72, 15]. These contexts usually lack an independent set of registers, and instead share registers with the primary register set and/or have private registers written by hardware. Threads begin execution in the contexts on a cache miss or, alternatively, by request from the primary thread [92].

Mowry and Ramkissoon investigate software-based multithreading on an architecture in which a cache miss causes the processor to branch to a predetermined user-level PC [52]. They set this PC to a light-weight context-switch routine and suggest compiler-based register-file partitioning to reduce context-switch overhead.

Multiple threads executing within a single stack frame and possibly a single register set have been investigated in the context of dataflow architectures on large parallel machines, such as *T, pRISC, and TAM [53, 54, 24]. In these architectures, threads consists of only a few instructions, and are used to hide latencies of, for example, memory operations. The compiler manages register and stack frame usage between threads. This management is typically conservative because of the large number of threads and the uncertainty in their dynamic execution order. For example, threads may not rely on any values in registers when they begin execution.

Every architecture designer must decide how many registers to support in the ISA. Researchers have investigated the sensitivity of applications to the number of architectural registers. Bradlee et al. [12] found that reducing the number of integer registers from 32 to 16 had a negligible effect on execution time on some integer programs, and caused a 17% degradation on scientific applications. Postiff et al. [61] argue that application sensitivity to the number of architectural registers increases as compiler technology improves.

$_{mt}$SMT conserves registers by mapping a single architectural register set among multiple mini-threads. Researchers have explored other ways to reduce the register file burden in architectures. Cruz et al. [23] suggest structuring the register file as a multi-level cache, complete with a pseudo-LRU replacement policy. They find that, due to the savings in access time, such an organization outperforms a non-pipelined, single banked architecture by 90% for the SPEC95 benchmarks. Monreal et al. [51] focus on conserving renaming registers by delaying the pipeline stage at which physical registers for destination operands are allocated. They find a 25% reduction in the number of renaming registers with little loss in performance. Lo et al. [47] investigate deallocating registers on SMT after their last use via compiler-inserted annotations. They observed up to an average speedup of 60% with the most efficient annotation mechanisms. They also found that deallocating the registers of idle contexts supports a 25% reduction in the number of registers on a 4-con-

text SMT with no loss in performance. Lo and Monreal both focus on improving the sharing of renaming registers. $_{mt}$SMT focuses on economizing architectural registers as well as renaming registers, and could work synergistically with both of their techniques. All three of these techniques could benefit from Cruz's optimizations.

5.6 Chapter summary

This chapter explored the final software interface issue, that of architectural register usage on SMT. Small-scale SMTs will clearly become a part of the CPU landscape in the next several years. While such CPUs can obtain significant performance improvements through multithreading, they are still likely to underutilize the massive processing and storage resources available on the next generation of out-of-order processors.

This chapter has introduced and evaluated a simple modification to SMT that greatly increases throughput. This modification, called mini-threads, adds partial thread-state hardware to each context, allowing mini-threads executing within the same context to *share* its architectural register set. Consequently, mini-threads allow processor implementors to increase the degree of parallelism supported by SMT by side-stepping a primary impediment to scaling up SMT: the size of the register file and renaming hardware. Essentially, they propel SMT further along the throughput curve.

Implementing mini-threads allows applications to trade off register set size for TLP. Each application decides independently whether or not to use mini-threads. If it ignores the mini-contexts, the machine behaves identically to an SMT. Alternatively, if it chooses to create mini-threads, it can boost TLP and machine throughput. However, since mini-threads in each context share the architectural register set, the application must arrange for its threads to manage it. Because of the flexibility of applications to use mini-threads only when beneficial, adding mini-thread contexts to SMT will never degrade performance for single-program workloads.

This chapter evaluates mini-threads with a statically partitioned register model and two and three mini-threads per context on the Apache web server and four SPLASH-2 parallel scientific applications. Overall, our results show that trading off register set size for TLP improves performance, especially on the smaller SMTs, which are characteristic

of the first commercial implementations. (Adding mini-threads improves performance over all applications evaluated by an average of 48% and a maximum of 88% on a 2-context SMT, with decreasing improvements on successively larger SMTs.) The reason for the large improvement is that almost all applications can exploit the extra mini-threads to boost IPC, and most suffer only minor performance degradations due to partitioning the register set. In particular, restricting applications to half or a third of the register set degraded average performance by only 5% and 15% respectively. However, the TLP improvement due to the extra mini-threads ranged from 60% on a small 2-context SMT to 7% on an 8-context SMT.

Chapter 6

Conclusion and future work

SMT has gradually progressed from a research idea to commercial processor technology. This thesis explored three software interface issues on SMT that are important to its real-world applicability. These issues are: operating system performance on SMT, the impact of spinning on SMT, and register file limitations to scaling SMT.

We investigated these issues with a new, detailed simulation infrastructure. This infrastructure combines an application-level SMT simulator with the SimOS machine simulation framework, allowing us to include all OS instructions and events (interrupts, privileged processor state, etc.) in our simulations. We discussed the challenges of developing this complex model and described other tools that proved invaluable in debugging the simulator and analyzing data. We showed that it is relatively straightforward to modify an SMP-aware operating system to execute on SMT.

The first software interface we explored was operating system execution. This thesis reported the first measurements of an OS executing on SMT. Our results show that, for the SPECInt95 workload, despite high kernel memory subsystem and branch prediction miss rates, SMT instruction throughput was perturbed only slightly by the operating system, since kernel activity in SPECInt programs is small and SMT hides latencies well. However, on a superscalar, including the operating system in simulations perturbed bottomline performance more than on an SMT (a 15% vs. a 5% drop in IPC), because key hardware resources (the instruction cache and the L2 cache) were stressed several-fold and superscalar performance is more susceptible to instruction latencies. These results suggest that researchers interested in bottomline performance for SPECInt-like scientific applications can confidently rely on application-level simulations on SMT, but should be less confident of omitting operating system effects when evaluating superscalar architectures.

In contrast, we found that the Apache web server spends most of its time in the operating system, executing file system and networking operations. The Apache OS-intensive workload is very stressful to a processor, causing significant increases in cache miss rates compared to SPECInt (e.g., the I-cache miss rate jumps from 2% to 5%). Inter-thread kernel interference occurs because SMT can execute instructions from multiple kernel threads simultaneously. As a result, Apache achieves an IPC of only 1.1 on a superscalar, compared to SPECInt's IPC of 2.6. SMT's latency tolerance is able to compensate for many of the demands of operating system code. When executing Apache, SMT achieves a 4-fold improvement in throughput over the superscalar, the highest relative gain of any SMT workload to date.

Second, we studied synchronization instructions on SMT. We examined the cost of spinning synchronization on SMT on various workloads and processor configurations. Spinning has the potential to be especially harmful on SMT because all executing threads share pipeline resources. To quantify the impact of spinning, we replaced two common OS spin loops, the idle loop and the simple lock-acquire spin loop, with non-spinning versions based on SMT hardware-blocking locks. We examined lock-acquire spinning with the Apache web server workload and idle spinning with a variety of multiprogrammed SPECInt95 workloads. Overall, we observed that spinning harms performance only to the extent that it consumes pipeline resources that non-spinning threads could have used effectively. For example, removing spinning from the Apache workload executing on an eight-context SMT improved performance by only 1%. This change is small because lock contention causes only minor spinning and a loaded, well-configured web server idles little. However, even on a version of the Apache workload modified to increase lock-acquire spinning to 50% of committed instructions, replacing spinning with SMT-based lock-acquire routines improved performance by only 13%, because Apache struggles to utilize the processor, despite its abundance of threads.

SPECInt95 applications better utilize the processor and so exhibit more sensitivity to spinning. In workloads of four SPECInt95 applications executing on an eight-context SMT, idle-loop spinning constituted 50% of committed instructions. Removing it improved performance by 75%, compared to only 14% for the Apache workload with a similar amount of spinning. Single thread performance, an important SMT design consid-

eration, suffers more from spinning. Removing idle-loop spinning improved performance by between 2x and 4x for workloads in which a single SPECInt95 application executes on an eight-context SMT.

The third issue we examined is architectural register sharing on SMT. We introduced and evaluated a simple modification to SMT that greatly increases instruction throughput. This modification, called mini-threads, adds partial thread-state hardware to each context, allowing mini-threads executing within the same context to *share* the context's architectural register set. Consequently, mini-threads allow processor implementors to increase the degree of parallelism supported by SMT by side-stepping a primary impediment to scaling up SMT: the size of and the time needed to access the register file and renaming hardware.

Implementing mini-threads allows applications to trade off architectural register set size for thread-level parallelism (TLP). Each application decides independently whether or not to use mini-threads. If it ignores the mini-contexts, the machine behaves identically to an SMT. Alternatively, if it chooses to create mini-threads, it can boost TLP and machine throughput. However, since mini-threads in each context share the architectural register set, the application must arrange for its threads to manage the shared registers. Because of the flexibility of applications to use mini-threads only when beneficial, adding mini-thread contexts to SMT will never degrade performance for single-program workloads.

We evaluated mini-threads with a statically partitioned register model and two and three mini-threads per context. Our workloads consisted of the Apache web server and four SPLASH-2 parallel scientific applications. Overall, our results show that trading off register set size for TLP improves performance, especially on the smaller SMTs, which are characteristic of the first commercial implementations. (Adding mini-threads improves performance over all applications evaluated by an average of 48% and a maximum of 88% on a 2-context SMT, with decreasing improvement on successively larger SMTs.) The reason for the large improvement is that almost all applications can exploit the extra mini-threads to boost IPC, and most suffer only minor performance degradations due to partitioning the register set among the mini-threads. In particular, restricting applications to half or a third of the register set degraded average performance by only 5% and 15%,

respectively. However, the TLP improvement due to the extra mini-threads ranged between 60% on a small 2-context SMT to 7% on an 8-context SMT.

6.1 Future work

We present four broad directions for future work. First, developing the simulator infrastructure raised interesting methodological questions. In addition, our investigation of each of the three software interface issues suggested natural extensions for future work.

Our detailed simulation infrastructure made analyzing pipeline simulation data challenging. Traditional techniques such as analyzing hit-rate metrics do not permit any comparison of relative impact. For example, how does one determine if a 10% cache miss ratio limits performance more than a 5% branch misprediction rate, or if either impacts processor throughput? We are developing a technique that expresses hardware component performance in common units. Using common units permits relative comparison of component performance. These units also provide an indication of how component performance affects bottom line processor throughput. A prototype of this technique proved invaluable in deciphering the simulation results used in this thesis. We intend to more thoroughly evaluate it and compare it to more traditional techniques.

Additional opportunity exists for OS optimizations specific to SMT beyond that which we explored. For example, tailoring OS data structures to SMT could improve both cache and synchronization performance. On an SMP, each processor contains a private cache and a coherence protocol maintains consistency between them. Using partitioned data structures can simultaneously improve cache and synchronization performance. However, on SMT, all executing threads share the caches. The shared caches introduce a unique trade-off in data structures between a centralized one to improve cache performance versus a partitioned one to reduce lock contention.

The minimal impact of heavy spinning on Apache performance suggests that a carefully constructed thread could accomplish substantial work with little impact on other executing threads (e.g., Apache threads). Understanding how to construct such a thread and how much work it can unobtrusively accomplish would aid in applications such as deciding when to execute the optimizing routines in the idle loop (discussed in Section 4.1.1),

building performance monitoring threads on SMT, or other situations involving a helper thread (Section 5.1.2 references some examples).

Finally, our mini-thread investigation raises the question of other ways to improve register file utilization. We focused on statically partitioning the register set equally among the mini-threads that share it. However, nothing in the mini-thread architecture precludes other more aggressive schemes. We see two directions in which to extend this work. The first improves the efficiency of the mapping of architectural registers to the needs of executing mini-threads. For example, static, unequal partitioning of the register set would allow a light-weight mini-thread with minimal register needs to efficiently execute with a register-hungry mini-thread. Alternatively, partitions whose size varies as mini-threads execute could track the dynamically changing register needs of mini-threads.

The second direction leverages architectural register sharing to share register values between mini-threads. Sharing values could reduce startup and communication costs between executing mini-threads, which may allow efficient parallelization to a finer degree than otherwise possible. Possible schemes for sharing register values mirrors those described above for partitioning. The register set could be divided, perhaps unequally, into registers that are shared between mini-threads and registers that are private to each mini-thread. Further, this mapping could change with the dynamic needs of mini-threads.

These aggressive architectural register usage schemes will demand more advanced compiler technology. The overhead of sharing values through registers must compete against communication through the L1 data cache that current architectures provide. In most architectures, each instruction explicitly names each architectural register, and this naming is determined statically at compile time. The ability to dynamically alter register usage may require novel compiler/architectural techniques.

References

[1] AGARWAL, A., HOROWITZ, M., AND HENNESSY, J. Cache performance of operating systems and multiprogramming workloads. *ACM Transactions on Computer Systems 6*, 4 (November 1988).

[2] AGARWAL, A., LIM, B.-H., KRANZ, D., AND KUBIATOWICZ, J. April: A processor architecture for multiprocessing. In *Proceedings of the International Symposium on Computer Architecture* (June 1990).

[3] AKKARY, H., AND DRISCOLL, M. A. A dynamic multithreading processor. In *Proceedings of the International Symposium on Microarchitecture* (November 1998).

[4] ALEMANY, J., AND FELTEN, E. W. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *Proceedings of the Symposium on Principles of Distributed Computing* (August 1992).

[5] ANDERSON, T. E. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems 1*, 1 (January 1990).

[6] ANDERSON, T. E., BERSHAD, B. N., LAZOWSKA, E. D., AND LEVY, H. M. Scheduler activations: effective kernel support for the user-level management of parallelism. In *Proceedings of the Symposium on Operating Systems Principles* (October 1991).

[7] APACHE SOFTWARE FOUNDATION. *Apache Web Server.* http://www.apache.org/.

[8] BALASUBRAMONIAN, R., DWARKADAS, S., AND ALBONESI, D. H. Dynamically allocating processor resources between nearby and distant ILP. In *Proceedings of the International Symposium on Computer Architecture* (June 2001).

[9] BARROSO, L., GHARACHORLOO, K., AND BUGNION, F. Memory system characterization of commercial workloads. In *Proceedings of the International Symposium on Computer Architecture* (June 1998).

[10] BERSHAD, B. N. Practical considerations for non-blocking concurrent objects. In *Proceedings of the International Conference on Distributed Computing Systems* (May 1993).

[11] BRADFORD, J. P., AND ABRAHAM, S. Efficient synchronization for multithreaded processors. In *Proceedings of the Workshop on Multithreaded Execution, Architecture and Compilation* (January 1998).

[12] BRADLEE, D., EGGERS, S., AND HENRY, R. The effect on RISC performance of register set size and structure versus code generation strategy. In *Proceedings of the International Symposium on Computer Architecture* (May 1991).

[13] BURNS, J., AND GAUDIOT, J.-L. Quantifying the SMT layout overhead - does SMT pull its weight? In *Proceedings of the International Symposium on High-Performance Computer Architecture* (January 2000).

[14] CHAPIN, S. Distributed and multiprocessor scheduling. *ACM Computer Surveys 28*, 1 (March 1996).

[15] CHAPPELL, R., STARK, J., KIM, S., REINHARDT, S., AND PATT, Y. Simultaneous subordinate microthreading (SSMT). In *Proceedings of the International Symposium on Computer Architecture* (May 1999).

[16] CHEN, J. B., AND BERSHAD, B. N. The impact of operating system structure on memory system performance. In *Proceedings of the Symposium on Operating Systems Principles* (December 1993).

[17] CLARK, D. W., AND EMER, J. S. Performance of the VAX-11/780 translation buffer : Simulation and measurement. *ACM Transactions on Computer Systems 3*, 1 (1985).

[18] COLLINS, J. D., WANG, H., TULLSEN, D. M., HUGHES, C., LEE, Y.-F., LAVERY, D., AND SHEN, J. P. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the International Symposium on Computer Architecture* (June 2001).

[19] COMPAQ. *Alpha 21264 Hardware Reference Manual.* http://www.support.compaq.com.

[20] COMPAQ. *ATOM.* http://www.tru64unix.compaq.com/developerstoolkit/#atom.

[21] COMPAQ. *SimOS-Alpha.* http://www.research.digital.com/wrl/projects/SimOS/.

[22] CROWLEY, P., FIUCZYNSKI, M. E., BAER, J., AND BERSHAD, B. N. Characterizing processor architectures for programmable network interfaces. In *Proceedings of the International Conference on Supercomputing* (May 2000).

[23] CRUZ, J.-L., GONZÁLEZ, A., VALERO, M., AND TOPHAM, N. P. Multiple-banked register file architectures. In *Proceedings of the International Symposium on Computer Architecture* (June 2000).

[24] CULLER, D. E., SAH, A., SCHAUSER, K. E., VON EICKEN, T., AND WAWRZYNEK, J. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (April 1991).

[25] EGGERS, S., EMER, J., LEVY, H., LO, J., AND STAMM, R. Simultaneous multithreading: A foundation for next-generation processors. In *Proceedings of the International Symposium on Microarchitecture* (August 1997).

[26] EICKEMEYER, R. J., JOHNSON, R. E., KUNKEL, S. R., SQUILLANTE, M. S., AND LIU, S. Evaluation of multithreaded uniprocessors for commercial application environments. In *Proceedings of the International Symposium on Computer Architecture* (May 1996).

[27] EMER, J. Ev8: The post-ultimate alpha. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (September 2001).

[28] FREE SOFTWARE FOUNDATION. *GNU gprof - Table of Contents.* http://www.gnu.org/manual/gprof-2.9.1/gprof.html.

[29] GLOY, N., YOUNG, C., CHEN, J. B., AND SMITH, M. D. An analysis of dynamic branch prediction schemes on system workloads. In *Proceedings of the International Symposium on Computer Architecture* (May 1996).

[30] GUPTA, A., HENNESSY, J., GHARACHORLOO, K., MOWRY, T., AND WEBER, W. Comparative evaluation of latency reducing and tolerating techniques. In *Proceedings of the International Symposium on Computer Architecture* (May 1991).

[31] GWENLAPP, L. Dansoft develops VLIW design. *Microprocessor Report 11*, 2 (February 1997).

[32] GWENNAP, L. MIPS R10000 uses decoupled architecture. *Microprocessor Report 8*, 14 (October 1994).

[33] GWENNAP, L. Digital 21264 sets new standard. *Microprocessor Report 10*, 14 (October 1996).

[34] HAMMOND, L., AND OLUKOTUN, K. Considerations in the design of Hydra: A multiprocessor-on-a-chip microarchitecture. Technical Report CSL-TR-98-749, Stanford University, Computer Systems Laboratory (February 1998).

[35] HAMMOND, L., WILLEY, M., AND OLUKOTUN, K. Data speculation support for a chip multiprocessor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1998).

[36] HERLIHY, M. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems 15*, 5 (November 1993).

[37] HERLIHY, M. P. Impossibility and universality results for walt-free synchronization. In *Proceedings of the Symposium on Principles of Distributed Computing* (August 1998).

[38] HU, Y., NANDA, A., AND YANG, Q. Measurement, analysis and performance improvement of the Apache web server. In *Proceedings of the IEEE International Performance, Computing and Communications Conference* (February 1999).

[39] INTEL. *Hyper-Threading Technology.* http://developer.intel.com/technology/hyperthread/.

[40] KARLIN, A. R., LI, K., MANASSE, M. S., AND OWICKI, S. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proceedings of the Symposium on Operating Systems Principles* (October 1991).

[41] LAUDON, J., GUPTA, A., AND HOROWITZ, M. Interleaving: A multithreading technique targeting multiprocessors and workstations. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1994).

[42] LIM, B.-H., AND AGARWAL, A. Waiting algorithms for synchronization in large-scale multiprocessors. *ACM Transactions on Computer Systems 11*, 3 (August 1993).

[43] LO, J., BARROSO, L., EGGERS, S., GHARACHORLOO, K., LEVY, J., AND PAREKH, S. An analysis of database workload performance on simultaneous multithreading processors. In *Proceedings of the International Symposium on Computer Architecture* (June 1998).

[44] LO, J., EGGERS, S., EMER, J., LEVY, H., STAMM, R., AND TULLSEN, D. Converting thread-level parallelism into instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems 15*, 2 (August 1997).

[45] LO, J., EGGERS, S., LEVY, H., PAREKH, S., AND TULLSEN, D. Tuning compiler optimizations for simultaneous multithreading. In *Proceedings of the International Symposium on Microarchitecture* (December 1997).

[46] LO, J., EGGERS, S., LEVY, H., AND TULLSEN, D. Compilation issues for a simultaneous multithreading processor. In *Proceedings of the SUIF Compiler Workshop* (January 1996).

[47] LO, J., PAREKH, S., EGGERS, S., LEVY, H., AND TULLSEN, D. Software-directed register deallocation for simultaneous multithreading processors. *IEEE Transactions on Parallel and Distributed Systems 10*, 9 (September 1999).

[48] MAYNARD, A., DONNELLY, C., AND OLSZEWSKI, B. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1994).

[49] MCFARLING, S. Combining branch predictors. Technical Report TN-36, Digital Equipment Corporation, Western Research Lab (June 1993).

[50] MELLOR-CRUMMEY, J. M., AND SCOTT, M. L. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems 9*, 1 (February 1991).

[51] MONREAL, T., GONZÁLEZ, A., VALERO, M., GONZÁLEZ, J., AND NALS, V. V. Delaying physical register allocation through virtual-physical registers. In *Proceedings of the International Symposium on Microarchitecture* (November 1999).

[52] MOWRY, T. C., AND RAMKISSOON, S. R. Software-controlled multithreading using informing memory operations. In *Proceedings of the International Symposium on High-Performance Computer Architecture* (January 2000).

[53] NIKHIL, R. S. Can dataflow subsume von neumann computing? In *Proceedings of the International Symposium on Computer Architecture* (June 1989).

[54] NIKHIL, R. S., PAPADOPOULOS, G. M., AND ARVIND. *T: A multithreaded massively parallel architecture. In *Proceedings of the International Symposium on Computer Architecture* (May 1992).

[55] OLUKOTUN, K., HAMMOND, L., AND WILLEY, M. Improving the performance of speculatively parallel applications on the Hydra CMP. In *Proceedings of the International Conference on Supercomputing* (June 1999).

[56] OLUKOTUN, K., NAYFEH, B. A., HAMMOND, L., WILSON, K., AND CHANG, K. The case for a single-chip multiprocessor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1996).

[57] OUSTERHOUT, J. K. Tcl: An embeddable command language. In *Proceedings of the USENIX Technical Conference* (January 1990).

[58] PAI, V., DRUSCHEL, P., AND ZWAENEPOEL, W. Flash: An efficient and portable web server. In *Proceedings of the USENIX Technical Conference* (June 1999).

[59] PAREKH, S., EGGERS, S., AND LEVY, H. Thread-sensitive scheduling for SMT processors. Technical report, Dept. of Computer Science, University of Washington (2000).

[60] PORTNOV, D. Evaluating SMT web server performance. Technical report, Dept. of Computer Science, University of Washington (February 2002).

[61] POSTIFF, M., GREENE, D., AND MUDGE, T. The need for large register files in integer codes. Technical Report CSE-TR-434-00, EECS/CSE University of Michigan (July 2000).

[62] PRESTON, R., BADEAU, R., BAILEY, D., BELL, S., BIRO, L., BOWHILL, W., DEVER, D., FELIX, S., GAMMACK, R., GERMINI, V., GOWAN, M., GRONOWSKI, P., JACKSON, D., MEHTA, S., MORTON, S., PICKHOLTZ, J., REILLY, M., AND SMITH, M. Design of an 8-wide superscalar RISC microprocessor with simultaneous multithreading. In *IEEE International Solid-State Circuits Conference Digest and Visuals Supplement* (February 2002).

[63] RADHAKRISHNAN, R., AND RAWSON, F. Characterizing the behavior of Windows NT web server workloads using processor performance counters. In *Proceedings of Workload Characterization: Methodology and Case Studies* (November 1999).

[64] REILLY, J. *SPEC describes SPEC95 products and benchmarks.* http://www.specbench.org/osg/cpu95/news/cpu95descr.html.

[65] ROSENBLUM, M., BUGNION, E., HERROD, S., WITCHEL, E., AND GUPTA, A. The impact of architectural trends on operating system performance. In *Proceedings of the Symposium on Operating Systems Principles* (December 1995).

[66] ROSENBLUM, M., HERROD, S. A., WITCHEL, E., AND GUPTA, A. Complete computer system simulation: The SimOS approach. *IEEE Parallel and Distributed Technology 3*, 4 (Winter 1995).

[67] ROTH, A., AND SOHI, G. Speculative data-driven multithreading. In *Proceedings of the International Symposium on High-Performance Computer Architecture* (January 2001).

[68] SIGMUND, U., AND UNGERER, T. Memory hierarchy studies of multimedia-enhanced simultaneous multithreaded processors for MPEG-2 video decompression. In *Proceedings of the Workshop on Multithreaded Execution, Architecture and Compilation* (August 2000).

[69] SITES, R. L. How to use 1000 registers. In *Proceedings of 1st Caltech Conference on VLSI* (January 1979).

[70] SNAVELY, A., AND TULLSEN, D. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (November 2000).

[71] SOHN, A., KODAMA, Y., KU, J., SATO, M., SAKANE, H., YAMANA, H., SAKAI, S., AND YAMAGUCHI, Y. Fine-grain multithreading with the EM-X multiprocessor. In *Proceedings of the Symposium on Parallel Algorithms and Architectures* (June 1997).

[72] SONG, Y., AND DUBOIS, M. Assisted execution. Technical Report CENG 98-25, Department of EE-Systems, University of Southern California (October 1998).

[73] SPECBENCH. *An explanation of the SPECWeb96 benchmark.* http://www.specbench.org/osg/web96/webpaper.html.

[74] STANFORD UNIVERSITY. *Stanford Parallel Applications for Shared Memory (SPLASH).* http://www-flash.stanford.edu/apps/SPLASH/.

[75] SUN. *Sun says UltraSparc V two chips in one.* http://news.com.com/2100-1001-271135.html?legacy=cnet.

[76] SUNDARAMOORTHY, K., PURSER, Z., AND ROTENBERG, E. Slipstream processors: Improving both performance and fault tolerance. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (November 2000).

[77] SWANSON, S., MCDOWELL, L., SWIFT, M., EGGERS, S., AND LEVY, H. An evaluation of speculative instruction execution on simultaneous multithreaded processors. Technical report, University of Washington (January 2002).

[78] TORRELLAS, J., GUPTA, A., AND HENNESSY, J. Characterizing the caching and synchronization performance of a multiprocessor operating system. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (September 1992).

[79] TORRELLAS, J., XIA, C., AND DAIGLE, R. Optimizing instruction cache performance for operating system intensive workloads. In *Proceedings of the International Symposium on High-Performance Computer Architecture* (January 1995).

[80] TULLSEN, D. *The SMTSIM Multithreading Simulator.* University of California, San Diego, http://www.cs.ucsd.edu/users/tullsen/smtsim.html.

[81] TULLSEN, D., EGGERS, S., EMER, J., LEVY, H., LO, J., AND STAMM, R. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the International Symposium on Computer Architecture* (May 1996).

[82] TULLSEN, D. M., EGGERS, S., AND LEVY, H. M. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the International Symposium on Computer Architecture* (June 1995).

[83] TULLSEN, D. M., LO, J. L., EGGERS, S. J., AND LEVY, H. M. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *Proceedings of the International Symposium on High-Performance Computer Architecture* (January 1999).

[84] UNRAU, R. C., KRIEGER, O., GAMSA, B., AND STUMM, M. Experiences with locking in a NUMA multiprocessor operating system kernel. In *Proceedings of the Symposium on Operating Systems Design and Implementation* (November 1994).

[85] VERGHESE, B., DEVINE, S., GUPTA, A., AND ROSEMBLUM, M. Operating system support for improving data locality on CC-NUMA compute servers. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1996).

[86] WALDSPURGER, C. A., AND WEIHL, W. E. Register relocation: flexible contexts for multithreading. In *Proceedings of the International Symposium on Computer Architecture* (May 1993).

[87] WALLACE, S., CALDER, B., AND TULLSEN, D. Threaded multiple path execution. In *Proceedings of the International Symposium on Computer Architecture* (June 1998).

[88] XIA, C., AND TORRELLAS, J. Improving the data cache performance of multiprocessor operating systems. In *Proceedings of the International Symposium on High-Performance Computer Architecture* (February 1996).

[89] ZAHORJAN, J., LAZOWSKA, E. D., AND EAGER, D. L. Spinning versus blocking in parallel systems with uncertainty. In *Proceedings of the International Symposium on the Performance of Distributed and Parallel Systems* (December 1988).

[90] ZAHORJAN, J., LAZOWSKA, E. D., AND EAGER, D. L. The effect of scheduling discipline on spin overhead in shared memory parallel systems. *IEEE Transactions on Parallel and Distributed Systems 2*, 2 (April 1991).

[91] ZILLES, C., EMER, J., AND SOHI, G. The use of multithreading for exception handling. In *Proceedings of the International Symposium on Microarchitecture* (November 1999).

[92] ZILLES, C., AND SOHI, G. Execution-based prediction using speculative slices. In *Proceedings of the International Symposium on Computer Architecture* (July 2001).

Vita

Joshua Abram Redstone was born on July 29, 1971 in Boston, Massachusetts. He graduated from Cornell University in 1993, receiving the B.A. degree (Magna cum Laude) in Computer Science. In 1996 he received the M.S. degree in Computer Science from the University of Washington, and completed his Ph.D. degree there in 2002.