Appeared in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

An Analysis of Operating System Behavior on a Simultaneous Multithreaded Architecture

Joshua A. Redstone, Susan J. Eggers and Henry M. Levy

University of Washington

{redstone, eggers, levy}@cs.washington.edu

Abstract

This paper presents the first analysis of operating system execution on a simultaneous multithreaded (SMT) processor. While SMT has been studied extensively over the past 6 years, previous research has focused entirely on user-mode execution. However, many of the applications most amenable to multithreading technologies spend a significant fraction of their time in kernel code. A full understanding of the behavior of such workloads therefore requires execution and measurement of the operating system, as well as the application itself.

To carry out this study, we (1) modified the Digital Unix 4.0d operating system to run on an SMT CPU, and (2) integrated our SMT Alpha instruction set simulator into the SimOS simulator to provide an execution environment. For an OS-intensive workload, we ran the multithreaded Apache Web server on an 8-context SMT. We compared Apache's user- and kernel-mode behavior to a standard multiprogrammed SPECInt workload, and compared the SMT processor to an out-of-order superscalar running both workloads. Overall, our results demonstrate the micro-architectural impact of an OS-intensive workload on an SMT processor and provide insight into the OS demands of the Apache Web server. The synergy between the SMT processor and Web and OS software produced a greater throughput gain over superscalar execution than seen on any previously examined workloads, including commercial databases and explicitly parallel programs.

1. INTRODUCTION

Simultaneous multithreading (SMT) is a latency-tolerant CPU architecture that executes multiple instructions from multiple threads each cycle. SMT works by converting thread-level parallelism into instruction-level parallelism, effectively feeding instructions from different threads into the functional units of a wide-issue, out-of-order superscalar processor [42, 41]. Over the last six years, SMT has been broadly studied [22, 23, 21, 45, 24, 43, 35] and Compaq has recently announced that the Alpha 21464 will include SMT [10]. 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.

This paper 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 percent of their execution time in the kernel [4], 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 systems 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.

For SMT, a functional processor and operating system do not yet exist. In lieu of these, we extended the SimOS-Alpha infrastructure [9], adding an Alpha-based SMT core as the instruction execution engine. SimOS [34] is a simulator detailed enough to boot and execute a complete operating system; in the case of the Compaq Alpha, SimOS executes PAL code as well. We also modified the Digital Unix 4.0d operating system to support SMT. This modification is quite straightforward, because Digital Unix is intended to run on conventional shared-memory multiprocessors and is therefore already synchronized for multithreaded operation.

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 [19], driven by the SPECWeb benchmark [32]. 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 paper is organized as follows. Section 2 details our measurement methodology, our simulation environment, and the workloads we use. Section 3 presents measurement results for our two workloads on SMT including operating system execution. The first half of Section 3 examines a multiprogrammed workload consisting of SPECInt applications, while the second half focuses on the Apache workload. Section 4 describes previous work and its relationship to our study, and we conclude in Section 5.

2. METHODOLOGY

This section describes the methodology used in our simulation-based experiments. We begin with a description of our SMT processor model and details of the simulated hardware configuration. We then describe the operating system simulation environment at both the hardware and software levels. Finally, we describe the two workloads evaluated: a multiprogramming workload of SPECInt95 benchmarks and the Apache Web server.

2.1 SMT and superscalar processor models

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 SMT as effective in increasing throughput on a variety of workloads, while still providing good performance for single-threaded applications [41, 22, 23, 21, 45].

At the hardware level, SMT is a straightforward extension of modern, out-of-order superscalars, such as the MIPS R10000 [15] or the Alpha 21264 [16]. 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. Compaq estimates that the modifications to an out-of-order superscalar necessary to support SMT translate into only a 10% increase in chip area [10].

Table 1 lists the parameters of the SMT processor and memory system simulated, chosen to be characteristic of processors in the near future. The out-of-order superscalar we evaluate is provisioned with hardware resources identical to the SMT, except that it lacks the extra hardware contexts and has 2 fewer pipeline stages, due to its smaller register file.

Pipeline	9 stages
Fetch Policy	8 instructions per cycle from up to 2 contexts (the 2.8 ICOUNT scheme of [41])
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 [26]
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
Cache Hierarchy	
Cache Line Size	64 bytes
Icache	128KB, 2-way set associative, single port
	2 cycle fill penalty
Dcache	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)
MSHR	32 entries for the L1 caches, 32 entries for the L2 cache
Store Buffer	32 entries
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

Table 1: SMT parameters.

2.2 Operating system execution

2.2.1 OS simulation environment

At one level the OS is simply a large program; however, it is unique 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). To simulate the OS thus requires simulating those resources and events. In this work, we built upon the SimOS-Alpha hardware simulation framework [9], integrating our SMT CPU simulator into SimOS. This allows us to boot and run the operating system on the simulator and include in our simulation every instruction, privileged or non-privileged, that would be executed on a real CPU. The SimOS environment also executes Alpha PAL code - a layer of software that exists below the operating system itself. PAL code is used, for example, to respond to TLB misses and to handle synchronization within the OS (SETIPL). We also model almost all OS/hardware interactions that affect the memory hierarchy, such as DMA operations and cache flush commands. The one exception is DMA operations from the network interface; although including network-related DMA would double the number of memory bus transactions for the Apache workload (the SPECInt workload doesn't use the network), the average memory bus delay would remain insignificant, since it is currently only 0.25 cycles per bus transaction.

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.

2.2.2 OS modifications

We execute the Compaq/Digital Unix 4.0d operating system, a (shared-memory) multiprocessor-aware OS. By allowing SMT to appear to the OS as a shared-memory multiprocessor (SMP), the only required changes to the OS occur where the SMT and SMP architectures differ. In the case of the Alpha, these differences are SMT's shared TLB and L1 caches, versus the per-processor TLB and L1 caches of an Alpha SMP. Of these two differences, only the TLB-related OS code required modification.

The Alpha TLB includes an address space number (ASN) tag on TLB entries, which allows multiple address spaces to 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.

Although the architectural interface to the caches differs between an SMT processor and an MP, this does not necessitate OS modifications. The interface provides commands to flush the L1 instruction and data caches, which in an SMT 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 is never incorrect.

The OS we execute contains the set of minimal changes required to run Digital Unix on an SMT, but does not explore the numerous opportunities for optimizations. For example, OS constructs such as the idle loop and spin locking are unnecessary and can waste resources on an SMT. (However, in the experiments presented in this paper, idle cycles constituted no more than 0.7% of steady-state CPU cycles, and spin locking accounted for less than 1.2% of the cycles in the SPECInt workload and less than 4.5% of cycles in the Apache workload.) Another possible optimization would be to replace the MP OS process scheduler with an SMT-optimized scheduler [36, 30]. We plan to investigate OS optimizations as future work, but it is encouraging that an SMP-aware OS can be modified in a straight-forward fashion to work on an SMT processor.

2.3 Simulated workloads

In this study, we examine 2 different workloads. The first is a multiprogrammed workload composed of all 8 applications from the SPEC95 integer suite [32], which we simulated for 650 million instructions. SPECInt95 was chosen for two reasons. First, since it is commonly used for architecture evaluations, including studies of SMT, we wished to understand what was omitted by not including OS activity in the previous work. Second, since Apache is also an integer program, the performance characteristics of SPECInt can serve as a baseline to help understand Apache's performance.

The second workload is Apache (version 1.3.4), a popular, public-domain Web server run by the majority of Web sites [19]. Because it makes heavy use of OS services (our measurements show that 75% of execution cycles are spent in the kernel), it is a rich environment in which to examine OS performance.[1] Most of the Apache data presented in this paper is based on simulations of over 1 billion instructions, starting at a point when the server is idle. However, the superscalar experiments in Section 3.2 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 [38]. We configured Apache with 64 server processes and SPECWeb with 128 clients that provide requests. To support the request rate needed to saturate Apache, we executed the SPECWeb benchmark as two driver processes, each with 64 clients. If the driver processes ran on a native Alpha (i.e., at full-speed), and drove our simulated Apache and OS software directly (SimOS causes slowdowns as much as a million-fold), then the network code would be unable to operate properly, and messages would be dropped by TCP. Therefore, we built a framework in which we run three copies of SimOS on a single Alpha: one executing Apache, and two running copies of SPECWeb96. The result is that the SPECWeb96 clients see exactly the same slowdown as Apache. The clients generate packets at a rate that can be handled by Apache, and the OS code on both sides can properly manage the network interface and protocols. Between the three SimOS environments, we simulate a direct network connection that transmits packets with no loss and no latency. The simulated network cards interrupt the CPUs at a time granularity of 10 ms, and the network simulator enforces a barrier synchronization across all machines every simulated 10 ms. The barrier keeps the simulators running in lock-step and guarantees deterministic execution of the simulations for repeatability of our experiments.

2.3.1 Simulating application code only

To more precisely characterize the impact of the OS on performance, we compared the simulation of a workload that includes the OS with one that models only application code. The application-only simulations are done with a separate simulator, derived from the SMT simulators that have been used in previous SMT studies. The application-only simulator models all system calls and kernel traps as completing instantly, with no effect on hardware state.

3. RESULTS

This section presents results from our SimOS-based measurements of operating system behavior and its impact on an SMT processor. In Section 3.1 we consider a SPECInt multiprogrammed workload; Section 3.2 examines an Apache workload and compares it to results seen for SPECInt.

3.1 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

[1] Apache was designed for portability. Its behavior may not be representative of other web servers, such as Flash [29], which were designed for performance (i.e., high throughput).

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 [13, 1]. Second, how does the impact of OS code on an 8-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, we examine similar questions in light of Apache, a more OS-intensive workload.

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

Figure 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 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 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. The majority of application-initiated system calls (Figure 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 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 2 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.

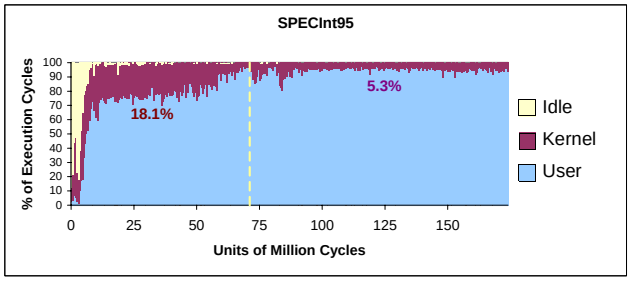


Figure 1. Breakdown of execution cycles when SPECInt95 executes on an SMT. Cycles spent in the kernel as a percentage of all execution cycles are shown as the dark color at the top.

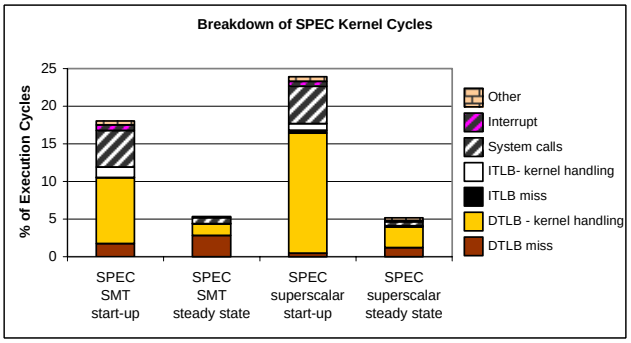


Figure 2. Breakdown of kernel time for SPECInt95.

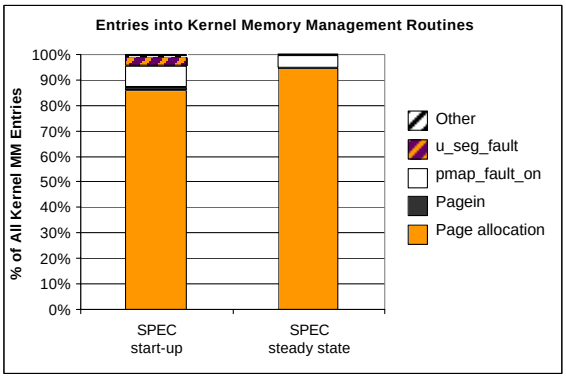


Figure 3. Incursions into kernel memory management code by number of entries.

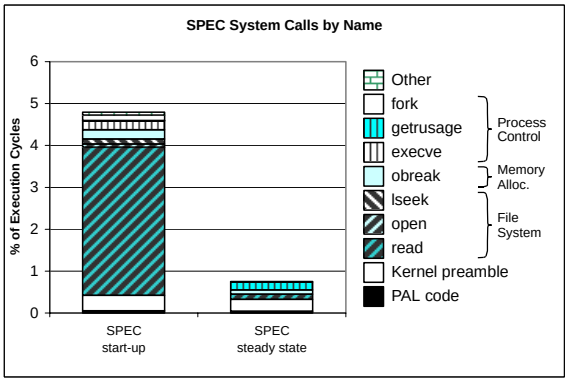


Figure 4. System calls as a percentage of total execution cycles.

Instruction Type	Program Start-up			Steady State		
	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

Table 2: 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.

	Miss Percentages									
	Branch Target Buffer		L1 Instruction Cache		L1 Data Cache		L2 Cache		Data TLB	
	User	Kernel	User	Kernel	User	Kernel	User	Kernel	User	Kernel
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%)**									
Intrathread conflicts	51.0	**4.9**	6.5	**.2**	14.6	**.8**	34.7	**1.2**	17.5	**.2**
Interthread 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 the OS			**40.9**	**10.7**						
Compulsory			.01	.01	7.3	.3	.5	36.6		

Table 3: The total miss rate and the 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).

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

Table 3 (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 [13, 1]. 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 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 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 4 (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

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 Icache miss rate (%)	1.0	2.0	190%	0.1	1.3	1300%
L1 Dcache 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%

Table 4: 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.

the most serious of the changes caused by simulating the kernel and depended on the interaction between two 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.

3.1.3 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 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 .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 4. 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 [41, 22, 21, 23], SMTs latency tolerance more than compensates for the additional interthread 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-behavior component.) This result suggests that researchers should be less confident of omitting effects of the operating system when evaluating superscalar architectures.

3.2 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.

3.2.1 The role of the operating system when executing Apache

Figure 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]

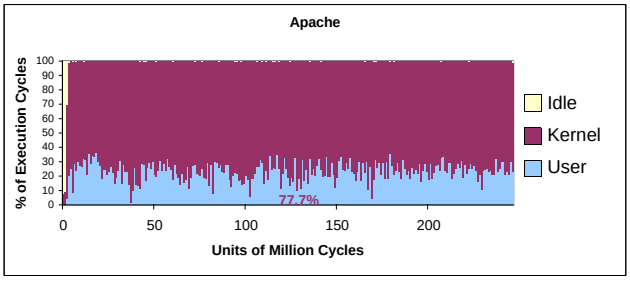


Figure 5. Kernel and user activity in Apache executing on an SMT.

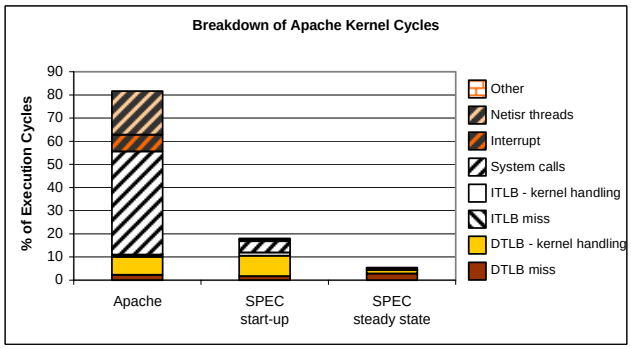
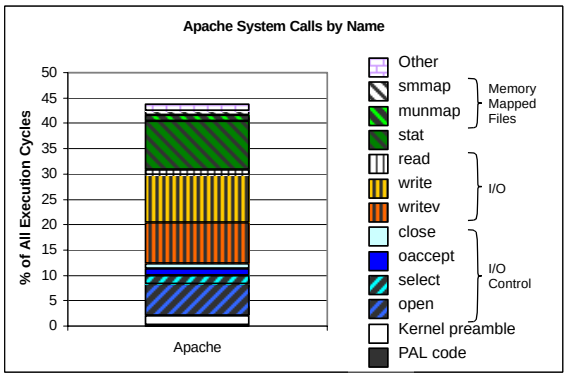


Figure 6. Breakdown of kernel activity in Apache on an SMT. Corresponding data from the startup and steady-state phases of the SPECInt workload are included for comparison.

Figure 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 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 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 processing 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 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

[1] [31] reported a similar ratio of user/kernel execution on a Pentium-based server.



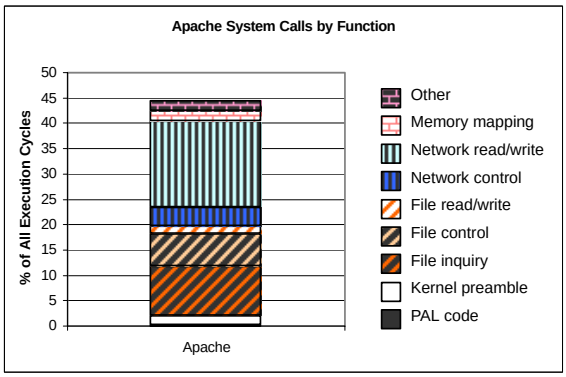


Figure 7. Breakdown of execution time spent processing kernel system calls on an 8-context SMT. The chart on the left shows the time spent in each system call. The chart on the right groups the system calls by activity. Note that the y-axis is the percentage of *all* execution cycles, not just kernel cycles.

services accounting for 21% of all kernel cycles and file services accounting for 18%.

3.2.2 Architectural performance characteristics

Table 5 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.

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 5: 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.

Table 6 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).

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 [11, 21]. 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

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 mispredict. rate (%)	9.1	9.3	7.4
ITLB miss rate (%)	.8	.0	.7
DTLB miss rate (%)	0.6	0.6	0.2
L1 Icache miss rate (%)	5.0	2.0	6.5
L1 Dcache 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$ misses	1.9	0.9	0.5
D$ misses	2.7	1.2	0.3
L2$ misses	1.3	1.0	0.2

Table 6: 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.

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.

3.2.3 Interthread competition and cooperation

As mentioned previously, SMT can issue instructions from *multiple* kernel threads in a single cycle, which creates new potentials for interthread conflicts. Table 7 presents more detail on the miss behavior of Apache, focusing on the causes for conflicts. Compared 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 Icache misses, 65% of L1 Dcache misses, and 41% of L2 cache misses are due to either intrathread or interthread kernel conflicts. These misses are roughly evenly split between the two categories, except in the L2 cache, where kernel interthread misses are almost twice as numerous as intrathread misses. User/kernel conflicts are very significant as well: 25% of L1 Icache misses, 10% of L1 Dcache 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 interthread conflicts are lower by 24%, 28%, and 38% for the Icache, Dcache, and L2 cache, respectively, when compared to Apache on an SMT.

In the BTB, kernel intrathread conflicts dominate, accounting for 68% of all BTB misses, while 6% of the misses are due to user/

	Percentage of Misses											
	Branch Target Buffer		L1 Instruction Cache		L1 Data Cache		L2 Cache		Data TLB		Instruction TLB	
	User	Kernel	User	Kernel	User	Kernel	User	Kernel	User	Kernel	User	Kernel
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	
Cause of the misses	**Percentage of Misses Due to Conflicts (sums to 100%)**											
Intrathread conflicts	12.2	**67.8**	6.4	**36.0**	5.8	**32.9**	.1	**13.9**	18.4	**7.3**	26.7	
Interthread 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 the OS			**.5**	**2.0**	**0**	**0**	**0**	**0**	**6.3**	**3.8**	**13.7**	
Compulsory			.1	.2	.3	6.9	2.0	31.8				

Table 7: 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).

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 interthread 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 8 shows, for several resources, the percentage of misses *avoided* due to constructive sharing in Apache. For example, on SMT, the overall miss rate of the L1 Icache would have been 66% higher, had it not been for Icache 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.

3.2.4 The effect of the operating system on hardware resources

Similar to the previous analysis of the SPECInt workload (Section 3.1.2 and Table 4), we now investigate the impact of the operating system on the cache and branch prediction hardware (Table 9[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 7, 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.

With the exception of the superscalar instruction cache miss rate, the OS had a greater effect on the hardware structures when

[1] Our simulators cannot execute Apache without operating systems code. However, we were able to omit operating systems references to the hardware components in Table 9, in order to capture user-only behavior.

Mode that would have missed	Misses avoided due to interthread prefetching as a percentage of total misses									
	Branch Target Buffer		L1 Instruction Cache		L1 Data Cache		L2 Cache		Data TLB	
	User	Kernel	User	Kernel	User	Kernel	User	Kernel	User	Kernel
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

Table 8: Percentage of misses avoided due to interthread 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 leftmost column would have encountered, if not for prefetching by other threads executing in the mode shown at the top of the column.

Metric	SMT			Superscalar		
	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 Icache miss rate (%)	0.9	5.0	5.5x	1.8	6.5	3.6x
L1 Dcache 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

Table 9: Impact of the operation system on specific hardware structures.

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).

3.3 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 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 interthread 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.

4. 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 [8] 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, Hennesy, 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 [7, 25] to obtain traces; while such instrumentation can capture all memory references, it perturbs workload execution [7]. Other studies employed bus monitors [12], 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 [5, 39, 46, 40]. As an example of more recent studies, Torrellas, Gupta, and Hennessy [39] 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 [25] looked at a trace-driven simulation of an IBM RISC system/6000 to investigate the performance of difference 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. [13] 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. [33], 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 [4] 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 [28, 44, 27, 17].

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 [19] 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 7. Radhakrishnan and Rawson [31] characterizes 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) [42, 41], SPLASH-2 [22], MPEG-2 decompression [35] and a database workload [21]. Evaluations of other multithreading and CMP architectures have similarly been limited to application code only [3, 18, 6, 2, 37, 20, 14] or PALcode [47].

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-mode code in the face of this low-level sharing. We also show how an operating-system intensive Web server workload benefits from simultaneous multithreading.

5. CONCLUSION

In this paper, 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 the 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 interthread 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.

Finally, we showed that it is relatively straightforward to modify an SMP-aware operating system to execute on a simultaneous multithreaded processor. In the future, we intend to experiment with OS structure in order to optimize the OS for the special features of SMT.

6. ACKNOWLEDGMENTS

This research was supported by the NSF grant MIP-9632977 and DARPA grant F30602-97-2-0226. The simulations were partially executed on Alpha systems donated by Compaq. The SMT simulator is based on a simulator originally written by Dean Tullsen and modified by Sujay Parekh. Manu Thambi created the SMT web server support. Marc Fiuczynski provided helpful discussions in the early stages of this work and the reviewers insightful comments on the submitted draft. We'd like to thank all for their support.

7. BIBLIOGRAPHY

[1] A. Agarwal, J. Hennessy, and M. Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Transactions on Computer Systems*, 6(4), May 1988.

[2] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. April: A processor architecture for multiprocessing. In *17th Annual International Symposium on Computer Architecture*, June 1990.

[3] H. Akkary and M. A. Driscoll. A dynamic multithreading processor. In *31st Annual International Symposium on Microarchitecture*, November 1998.

[4] L. Barroso, K. Gharachorloo, and E. Bugnion. Memory system characterization of commercial workloads. In *25nd Annual International Symposium on Computer Architecture*, July 1998.

[5] S. Chapin. Distributed and multiprocessor scheduling. *ACM Computing Surveys*, 28(1), March 1996.

[6] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous subordinate microthreading (SSMT). In *26th Annual International Symposium on Computer Architecture*, May 1999.

[7] J. B. Chen and B. N. Bershad. The impact of operating system structure on memory system performance. In *Symposium on Operating Systems Principals*, December 1993.

[8] D. Clark and J. Emer. Measurement of the VAX-11/780 translation buffer: Simulation and measurement. *ACM Transactions on Computer Systems*, 3(1), February 1985.

[9] Compaq, http://www.research.digital.com/wrl/projects/SimOS/. *SimOS-Alpha*.

[10] K. Diefendorff. Compaq chooses SMT for alpha. *Microprocessor Report*, 13(16), December 6 1999.

[11] S. Eggers, J. Emer, H. Levy, J. Lo, R. Stamm, and D. Tullsen. Simultaneous multithreading: A foundation for next-generation processors. *IEEE Micro*, 17(5), August 1997.

[12] R. J. Eickemeyer, R. E. Johnson, S. R. Kunkel, M. S. Squillante, and S. Liu. Evaluation of multithreaded uniprocessors for commercial application environments. In *23nd Annual International Symposium on Computer Architecture*, May 1996.

[13] N. Gloy, C. Young, J. Chen, and M. Smith. An analysis of dynamic branch prediction schemes on system workloads. In *23nd Annual International Symposium on Computer Architecture*, May 1996.

[14] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W. Weber. Comparative evaluation of latency reducing and tolerating techniques. In *18th Annual International Symposium on Computer Architecture*, May 1991.

[15] L. Gwennap. MIPS R10000 uses decoupled architecture, October 24 1994.

[16] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, 10(14), October 28 1996.

[17] L. Hammond and K. Olukotun. 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.

[18] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *8th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[19] Y. Hu, A. Nanda, and Q. Yang. Measurement, analysis and performance improvement of the apache web server. In *Proceedings of the 18th International Performance, Computing and Communications Conference*, February 1999.

[20] J. Laudon, A. Gupta, and M. Horowitz. Interleaving: A multithreading technique targeting multiprocessors and workstations. In *6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[21] J. Lo, L. Barroso, S. Eggers, K. Gharachorloo, J. Levy, and S. Parekh. An analysis of database workload performance on simultaneous multithreading processors. In *25nd Annual International Symposium on Computer Architecture*, June 1998.

[22] J. Lo, S. Eggers, J. Emer, H. Levy, R. Stamm, and D. Tullsen. Converting thread-level parallelism into instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, 15(2), August 1997.

[23] J. Lo, S. Eggers, H. Levy, S. Parekh, and D. Tullsen. Tuning compiler optimizations for simultaneous multithreading. In *30th Annual International Symposium on Microarchitecture*, December 1997.

[24] J. Lo, S. Parekh, S. Eggers, H. Levy, and D. Tullsen. Software-directed register deallocation for simultaneous multithreading processors. *IEEE Transactions on Parallel and Distributed Systems*, 10(9), September 1999.

[25] A. Maynard, C. Donnelly, and B. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[26] S. McFarling. Combining branch predictors. Technical report, TN-36, DEC-WRL, June 1993.

[27] K. Olukotun, L. Hammond, and M. Willey. Improving the performance of speculatively parallel applications on the Hydra CMP. In *International Conference on Supercomputing*, June 1999.

[28] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[29] V. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *USENIX Technical Conference*, June 1999.

[30] S. Parekh, S. Eggers, and H. Levy. Thread-sensitive scheduling for smt processors. Technical report, Department of Computer Science & Engineering, University of Washington, 2000.

[31] R. Radhakrishnan and F. Rawson. Characterizing the behavior of Windows NT web server workloads using processor performance counters. In *First Workshop on Workload Characterization*, November 1999.

[32] J. Reilly. SPEC describes SPEC95 products and benchmarks. September 1995. http://www.specbench.org/.

[33] M. Rosenblum, E. Bugnion, S. Herrod, E. Witchel, and A. Gupta. The impact of architectural trends on operating system performance. In *Symposium on Operating Systems Principals*, December 1995.

[34] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete computer simulation: The SimOS approach. *IEEE Parallel and Distributed Technology*, 3(4), Winter 1995.

[35] U. Sigmund and T. Ungerer. Memory hierarchy studies of multimedia-enhanced simultaneous multithreaded processors for MPEC-2 video decompression. In *Workshop on Multi-Threaded Execution, Architecture and Compilation*, January 2000.

[36] A. Snavely and D. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading architecture. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[37] A. Sohn, Y. Kodama, J. Ku, M. Sato, H. Sakane, H. Yamana, S. Sakai, and Y. Yamaguchi. Fine-grain multithreading with the EM-X multiprocessor. In *9th Annual Symposium on Parallel Algorithms and Architectures*, June 1997.

[38] SPECWeb. An explanation of the SPECWeb96 benchmark, 1996. http://www.specbench.org/.

[39] J. Torrellas, A. Gupta, and J. Hennessy. Characterizing the caching and synchronization performance of a multiprocessor operating system. In *5th International Conference on Architectural Support for Programming Languages and Operating Systems*, September 1992.

[40] J. Torrellas, C. Xia, and R. Daigle. Optimizing instruction cache performance for operating system intensive workloads. In *5th International Conference on Architectural Support for Programming Languages and Operating Systems*, January 1995.

[41] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23nd Annual International Symposium on Computer Architecture*, May 1996.

[42] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

[43] D. Tullsen, J. Lo, S. Eggers, and H. Levy. Supporting fine-grain synchronization on a simultaneous multithreaded processor. In *Fifth International Conference on High-Performance Computer Architecture*, January 1999.

[44] B. Verghese, S. Devine, A. Gupta, and M. Rosemblum. Operating system support for improving data locality on CC-NUMA compute servers. In *7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[45] S. Wallace, B. Calder, and D. Tullsen. Threaded multiple path execution. In *25nd Annual International Symposium on Computer Architecture*, June 1998.

[46] C. Xia and J. Torrellas. Improving the data cache performance of multiprocessor operating systems. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture)*, February 1996.

[47] C. Zilles, J. Emer, and G. Sohi. The use of multithreading for exception handling. In *32nd Annual International Symposium on Microarchitecture*, November 1999.