Thread-Sensitive Scheduling for SMT Processors

Sujay Parekh
IBM T.J. Watson Research Center
sujay@us.ibm.com

Susan Eggers
University of Washington
eggers@cs.washington.edu

Henry Levy
University of Washington
levy@cs.washington.edu

Jack Lo
Transmeta
jlo@transmeta.com

Abstract

A simultaneous-multithreaded (SMT) processor executes multiple instructions from multiple threads every cycle. As a result, threads on SMT processors – unlike those on traditional shared-memory machines – simultaneously share all low-level hardware resources in a single CPU. Because of this fine-grained resource sharing, SMT threads have the ability to interfere or conflict with each other, as well as to share these resources to mutual benefit.

This paper examines *thread-sensitive scheduling* for SMT processors. When more threads exist than hardware execution contexts, the operating system is responsible for selecting which threads to execute at any instant, inherently deciding which threads will compete for resources. Thread-sensitive scheduling uses thread-behavior feedback to choose the best set of threads to execute together, in order to maximize processor throughput. We introduce several thread-sensitive scheduling schemes and compare them to traditional oblivious schemes, such as round-robin. Our measurements show how these scheduling algorithms impact performance and the utilization of low-level hardware resources. We also demonstrate how thread-sensitive scheduling algorithms can be tuned to trade-off performance and fairness. For the workloads we measured, we show that an IPC-based thread-sensitive scheduling algorithm can achieve speedups over oblivious schemes of 7% to 15%, with minimal hardware costs.

1 Introduction

Simultaneous Multithreading (SMT) [22] is a processor design that combines the wide-issue capabilities of modern superscalars with the latency-hiding abilities of hardware multithreading. Using multiple on-chip thread contexts, an SMT processor issues instructions from multiple threads each cycle. The technique has been shown to boost processor utilization for wide-issue CPUs, achieving a 2- to 3-fold throughput improvement over conventional superscalars and a 2x improvement over fine-grained multithreading [10].

SMT is unique in the level of fine-grained resource sharing it permits. Because instructions from several threads execute simultaneously, threads compete every cycle for all common hardware resources, such as functional units, instruction queues, renaming registers, caches, and TLBs. Since programs may differ widely in their hardware requirements, some programs may interact poorly when co-scheduled onto the processor. For example, two programs with large cache footprints may cause inter-thread cache misses, leading to low instruction throughput for the machine as a whole. Conversely, threads with complementary resource requirements may coexist on the processor without excessive interference, thereby increasing utilization; for example, integer-intensive and FP-intensive benchmarks should execute well together, since they utilize different functional units. Consequently, thread scheduling decisions have the potential to affect performance, either improving it by co-scheduling threads with complementary hardware requirements, or degrading it by co-scheduling threads with identical hardware needs.

This paper presents and evaluates two classes of scheduling algorithms for SMTs: *oblivious algorithms*, which schedule without regard to thread behavior, and *thread-sensitive algorithms*, which predict and exploit the resource requirements of individual threads in order to increase performance. We first compare several oblivious schemes that differ in the number of context switches each quantum; we show that context switching alone is not a factor in scheduler performance. We then evaluate several thread-sensitive schemes that either target overall performance (IPC), focus on optimizing a single resource (such as the L1 D-cache, L2 cache, and TLB), or strive to utilize hardware resources in a complementary fashion. Our results show that a feedback-scheduler based on IPC is superior to the schemes that target a single hardware resource, achieving speedups over oblivious round-robin scheduling of 7% to 15% on the configurations and workloads we measured. Although the resource-specific schedulers improve the behavior of their particular resource, in doing so they expose other resource bottlenecks that then become dominant factors in constraining performance. We also consider thread starvation, and show how performance and fairness can be balanced in a system using thread-sensitive scheduling.

This paper is organized as follows. The next section describes previous work related to our study. Section 3 presents a brief overview of the SMT architecture and our simulator. In Section 4 we discuss the issues relevant to scheduling on SMT. We first evaluate the deficiencies of several simple thread-oblivious thread-scheduling algorithms; then using this information, we design and evaluate thread-sensitive algorithms that attempt to maximize the potential benefits obtained by SMT. Section 5 discusses the issue of scheduler fairness, and we conclude in Section 6.

2 Related Work

Previous studies of multithreaded machines either do not consider more threads than hardware contexts [18,2,10,1], or they use a simple round-robin scheme for scheduling threads onto the processor [13,6,14]. Fiske's thesis [11] looks at improving the performance of a single multithreaded application on a fine-grained multithreaded processor. He considers both mechanisms and policies for prioritizing various threads of the same application to meet different scheduling criteria. A multiprogrammed workload, however, is not discussed. The Tera processor scheduler [3] follows a Unix-like scheme for scheduling single-threaded programs. Both Unix [4,16] and Mach [5] use multi-level, priority-based feedback scheduling, which essentially amounts to round-robin for the compute-intensive workloads that we consider in this paper. They do not address the specific issue of selecting threads to improve processor utilization, which we consider here; their emphasis is more towards maintaining fairness.

Schedulers for multiprocessors also are faced with the problem of choosing the proper subset of threads to be active at a given moment. Typically, such schedulers focus on the issues of load balancing and cache affinity. Load balancing [8,20] assigns threads to processors so as to ensure that each processor is assigned an equivalent amount of work. If the load becomes unbalanced, a scheduler can move threads from one processor to another to help rebalance the load. However, relocating threads has a cost: the state built up by a thread in a processor's local cache is lost. In cache affinity scheduling [7,23,19], a thread is preferentially re-scheduled onto the processor on which it last executed, thus taking advantage of built-up cache state.

The simplest approach to adapting a traditional multiprocessor scheduler for an SMT CPU would equate each processor to a hardware thread context. However, using multiprocessor scheduling algorithms on SMT may not be appropriate. Load balancing, for example, is not a concern for a multithreaded machine, since hardware resources are dynamically shared by all threads; that is, one hardware context is as good as another. Cache affinity may also be less important, because there is only one shared cache, rather than several processor-private caches. On the other hand, keeping a thread loaded on an SMT processor may take advantage of its existing cache state and thus still be beneficial.

3 The SMT Model

This section briefly describes the SMT architecture and our SMT simulator. In general, we use the term *thread* to refer to a schedulable sequential task (an independent program or a component of a parallel program) and the word *context* to refer to the hardware structures that hold an executing thread's processor state. At any instant, a context holds a thread, and there are more threads than contexts.

An SMT processor executes multiple instructions from multiple threads each cycle. Even with this increased functionality, SMT can be constructed with straightforward modifications to a standard dynamically-scheduled superscalar processor [21]. To support multiple resident threads, several structures must be replicated or modified. These include per-context thread state (registers and a program counter), active lists, and mechanisms for pipeline flushing, traps, interrupts and return stack prediction. In addition, the branch-target buffer and TLB entries must include thread IDs. Modifications of this nature increased the chip area of Compaq's recently announced Alpha implementation of SMT by only 10%, compared to a similar superscalar design [9].

On an SMT processor, many processor structures are shared dynamically between executing threads. These include the renaming register pool, instruction queues, functional units, TLBs, and the entire memory hierarchy. It is contention for these resources that we need to consider when scheduling threads.

Table 1 lists the processor and memory system parameters for our SMT simulator which models a high-performance processor in the 3-year time frame. We simulate a processor pipeline similar to that of the Alpha 21264 [12]. Each cycle, the simulator fetches up to 8 instructions from each of 2 contexts. After instruction decode, register renaming maps per-context architectural registers onto a shared pool of physical registers. This not only removes false register dependencies within a thread (as in a conventional processor), but also resolves *interthread* naming conflicts. Renamed instructions reside in separate integer and floating-point queues, which are shared among the contexts. After their operands have been computed, instructions become eligible for issue to the functional units; ready instructions from any thread may issue any cycle. Finally, completed instructions are retired in per-thread program order.

For our experiments, we use cycle-accurate, emulation-based simulation of user-level instructions. Our simulator takes as input unmodified Alpha binaries and models the processor pipeline in detail, including speculation and resource contention. We also use a detailed TLB and cache simulator that models the entire memory hierarchy,

Hardware contexts	8 (occasionally 4 & 2 are examined for comparison)
Functional units	6 integer (4 ld/st), 4 FP
Instruction latencies	based on the Alpha 21264 [12]
Instruction queue entries	32 integer, 32 FP
Active list entries	128 entries/context
Architectural registers	32 integer, 32 FP (64-bit)
Renaming registers	100 integer, 100 FP
Instruction retirement	12 instructions/cycle
Branch prediction	McFarling-style [15]
Pattern history table	2K-entry
L1 cache organization (I & D)	128KB, 2-way, nonblocking; separate instruction, data
L2 cache organization	16MB, direct-mapped, unified
Fill latency	2 cycles (L1), 4 cycles (L2)
TBLs (I & D)	128 entries
Banks	4 (L1), 1 (L2)
Ports/bank	1 (L1-instruction and L2), 2 (L1-data)

Table 1: Processor and cache parameters

including bank and bus conflicts. We do not simulate the full operating system; instead, the simulator directly performs relevant functions that normally would be performed by the OS, such as I/O, page mapping and scheduling.

4 OS Thread Scheduling

When there are more threads than hardware contexts, the operating system must decide which subset of the runnable threads to schedule onto an SMT processor's hardware contexts. Thread scheduling can have several possible goals, e.g., optimizing response time, throughput, or processor utilization. Since the objective of SMT is to improve processor utilization by exploiting multiple threads, we will study scheduling algorithms that focus on this goal.

Because of the varying characteristics of different programs or phases within a single program, different combinations of threads can have varying resource demands. Hence, the particular thread mix chosen by the scheduler may impact processor utilization. In this paper, we investigate scheduling algorithms that rely on information about each program's behavior in order to make scheduling decisions. We characterize schedulers as *thread-sensitive* or *oblivious* according to whether or not they utilize thread-behavior information. We propose several thread-sensitive schedulers, each of which targets a different processor resource for optimization.

In the next few sections, we first present the methodology we use for carrying out our studies, including our evaluation procedure. We then present and analyze the performance of several oblivious scheduling algorithms. Based on this data, we create and analyze the thread-sensitive algorithms and compare them to the oblivious schemes.

4.1 Methodology

4.1.1 Workload

Our workload consists of a collection of single-threaded programs that are multiprogrammed onto the SMT processor. This scenario represents a heavily-loaded workstation or server that is running many different tasks at once. For our experiments, we use a mix of 32 programs from the SPEC95 [17] suite, 8 from SPEC-Int and 8 from SPEC-FP, each running on its reference dataset. (The specific programs are discussed later in Table 2.) We eliminated parallel threads from the study, because parallel threads typically execute the same code over different portions of a shared data space, and hence their per-thread hardware resource requirements are nearly identical.

Running any one of these SPEC applications with its reference dataset directly (i.e., not in the simulator) takes several minutes of CPU time. Running a multiprogramming workload in our detailed simulator would require many weeks for each run. To shorten the simulation time and still obtain realistic performance estimates, we retain the reference dataset but shorten the execution. We chose a window of execution of 100 million instructions[1] from each program as a representative sample of its execution. For the SPEC-int programs, this window is the first 100 million instructions. For SPEC-FP, the window begins in the main loop of the programs (the initial portion of these FP programs consists of either reading in or generating the input data). For correctness of both execution and cache contents, we execute the initialization portion of these benchmarks, but we start our measurements once the programs are past this (fast-forward) stage.

4.1.2 Simulations

Comparing alternative scheduling algorithms in an SMT environment is quite challenging, since experimental edge effects can potentially lead to erroneous conclusions. On the one hand, if a job mix is examined only in steady state (i.e., when all threads are still running), high throughput may be achieved during measurement in exchange for low throughput at a later time. For example, imagine an algorithm that chooses to execute only “good” threads and never executes the “bad” ones during an experiment. On the other hand, if a job mix is run until all threads complete, then throughput will inevitably drop at the tail of the experiment, when there are fewer threads than SMT hardware contexts. Depending on the relative lengths of steady-state and the tail, this low throughput may unrealistically dominate the experimental results.

For this reason, we investigated two different multiprogramming scenarios, each modeling a different set of workload and measurement conditions. The first scenario models SMT’s projected mode of execution, that of steady state in a heavily used compute, file, or web server. In this case, we assume a constant supply of threads for execution (e.g., a continuous stream of server requests, in which completing threads are replaced by new ones, and all contexts are always utilized). In the second scenario, we model a fixed multiprogramming workload that runs to completion. To examine the differences in these two scenarios, we run basically similar experiments, but we vary the measurement

1. To be precise, these are the retired instructions. The wrong-path instructions will vary in number and content.

interval and, in particular, the termination point for the measurements.

To represent the steady state server environment, we perform two measurement experiments. In SMTSteadyState, the processor executes until the first of the 32 schedulable threads terminates. Although, in practice, the supply of threads might be significantly larger, 32 is the maximum number of threads we could comfortably simulate. In the second experiment, 8ThreadsLeft, the processor executes until the pool of runnable threads is less than or equal to the number of SMT hardware contexts, i.e., there is no more work for the scheduler.

To study the run-to-completion effects, we performed two experiments as well. In general, we cannot know how much steady state activity precedes the tailing off of threads in a terminating workload. The longer the steady state period, the less evident will be the effect of the tail, and vice versa. Therefore, we gathered separate statistics for the tail, starting from the end of steady state, and we chose two different start-of-tail points. The 8Tail measurements contain instructions from the tail portion of 8ThreadsLeft; it begins when eight threads are left and tapers off to zero. SSTail is the remainder of SMTSteadyState; it begins with 31 threads (after the first thread completes) and also runs until all threads have finished. In addition, we ran to completion our entire 32-thread workload (called RunTo-Completion), capturing the effect of tailing off on one particular workload, that of 100 million instructions from each of 32 threads.

The various measurement scenarios described above were necessary to ensure that neither including nor ignoring scheduling edge effects would cause erroneous conclusions. In the end, however, we found that these differences in experimental and measurement conditions did not make an appreciable difference in the results. Therefore, in the sections below, we provide data for only a subset of the conditions that we measured, those that model steady state.

All simulations use a simple temporal job profile where we submit all the threads simultaneously at the start of the simulation. Some of the scheduling algorithms we discuss (e.g., round-robin) may be affected by the order in which the threads are listed. To counteract this effect, we tested several different initial orderings.

Our schedulers use a scheduling quantum of 1 million cycles[1] (about 2ms on a 500MHz processor). At the end of every quantum, the scheduling algorithm reevaluates the currently-loaded threads, selecting those threads that will execute for the next quantum. The scheduler may choose to replace all, some, or none of the threads from the previous quantum. All the schedulers we study allocate all hardware contexts if sufficient runnable threads are available. We assume that all threads are running at equal user priority.

4.1.3 Evaluation

The schedulers we study strive to improve processor utilization, which is normally measured by instructions per cycle (IPC). The problem with comparing the performance of different schedulers up to any intermediate point of our work-

1. We also experimented with quanta of 100K and 5M cycles. The results are qualitatively the same across all quantum values.

load (i.e., before workload completion) by their IPC is that, depending on the scheduler, the machine may execute a different instruction mix. This amounts to measuring performance across different workloads. In particular, a scheduler may execute the "easy" threads first and thus appear to be better than it is. To avoid this bias, we judge a scheduler's performance by its IPC improvement normalized to the single-threaded execution of *exactly* those instructions that were executed under the scheduler. To be precise:

1. Using a multi-threaded workload with a particular scheduler S, we measured:
   $\omega_S(p)$ = the number of instructions completed by thread p at the cutoff point under S
   Γ_S = the number of cycles at the cutoff point.
2. We ran each thread by itself to gather single-threaded baseline data of the form:
   $c(p,w)$ = the number of cycles for thread p to finish the first w instructions.
3. Then, scheduler performance was measured as the speedup over the single-threaded baseline:

$$\text{Speedup}_S = \frac{\sum_p c(p, \omega_S(p))}{\Gamma_S}$$

The weighted sum in the numerator gives the single-threaded baseline time, taking into account how much of each thread was executed in the multi-threaded workload. By normalizing to the single-threaded case, we avoid the problem caused by the cutoff. Even though they may have executed different instructions, schedulers can still be meaningfully compared when using these speedup values.

4.2 Oblivious Scheduling

We first present oblivious schedulers that do not make use of thread-specific information. In the descriptions below, we use n to refer to the total number of software threads and k for the number of hardware contexts.

- Round-robin (RR)
  The n threads are placed in a circular queue in the order submitted. The scheduler chooses k scheduled threads consecutively from this queue, effectively forming a window onto the queue. Every scheduling quantum, the window is shifted down by k. As threads finish, they are removed from the queue. We use Round-robin (RR) as a baseline for our comparisons to the thread-sensitive schedulers, based on its prevalent use, as discussed in Section 2.

- RR-by-1
  Context-switching in RR causes the entire set of k running threads to change each quantum. This can potentially destroy the working sets in the cache. RR-by-1 is a variant of RR that attempts to preserve temporal cache affinity by retaining k-1 of the threads from the previous quantum. That is, the window for RR-by-1 is shifted by just one every quantum.

- Random

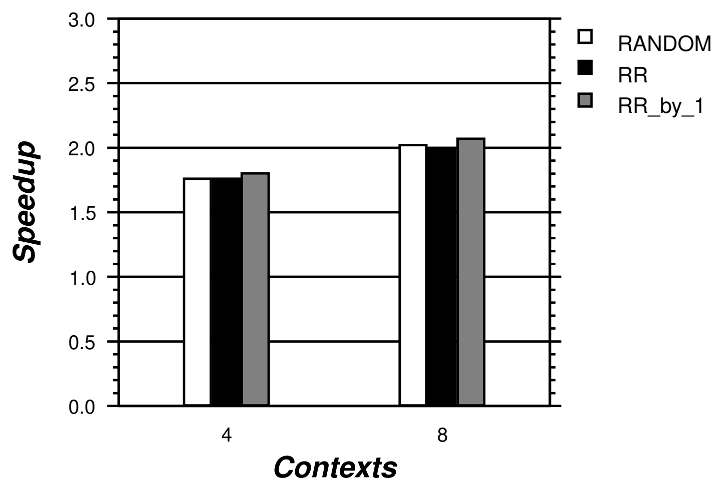

FIGURE 1. Speedup of the oblivious schedulers

Every scheduling interval, the *k* threads for the next interval are chosen randomly from all runnable threads and context switches are performed as required. Random enables us to judge whether any systematic bias is introduced by the grouping of threads in RR.

4.2.1 Performance of Oblivious Schedulers

Figure 1 shows the speedup achieved by the 3 oblivious schedulers for 4- and 8-context SMTs executing the 8ThreadsLeft workload (figures for SMTSteadyState are comparable). We can see that all of the oblivious schedulers achieve similar instruction throughput. There are two ramifications of this observation. First, the fact that Random does as well (or as poorly) as RR indicates that the performance of RR is not due to any systematic bias in the grouping of threads in the input program ordering. Second, RR-by-1 shows only marginally improved performance over RR, in spite of preserving cache state across scheduler invocations. This indicates that there may not be much benefit to preserving cache affinity on an SMT processor. We will elaborate on this point further in section 4.4.

To understand why the oblivious schedulers do not achieve better throughput, we examined in more detail the usage of several shared processor resources. For each resource of interest, we assumed perfect performance (or infinite size) and compared its performance to RR with the normal SMT configuration. Figure 2 shows the results, which indicate the potential for performance improvement by alleviating contention for a particular resource. (We use IPC here, since we are comparing alternatives with the same scheduling policy; IPC also lets us gauge the processor utilization achieved with RR. Again, the results are for the 8ThreadsLeft workload; figures for SMTSteadyState have on average an IPC 0.15 higher and retain the same relative performance across configurations.)

From the figure, we see that when hardware bottlenecks are removed, performance rises, in most cases considerably. Results for individual hardware resources depend on the number of concurrent threads, the number of inter-thread conflicts they generate, and the penalty of resolving those conflicts. With fewer thread contexts, the L1 D-cache is the largest single limiting resource. In this case, the small number of threads do not provide enough independent instruc-

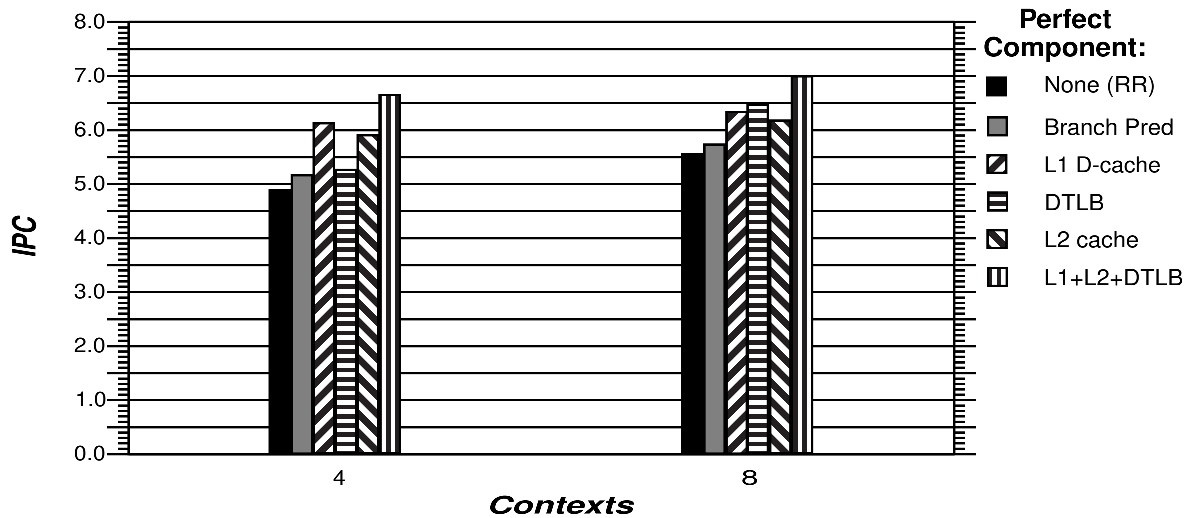

FIGURE 2. Effect of perfect components on commit IPC

tions for SMT to hide the inter-thread conflict miss latencies. Although with more threads, these misses continue to rise, the additional instructions allow SMT to better hide their latencies, and the L1 D-cache becomes less important.

This effect doesn't hold for the D-TLB, however, because the *relative* rise in TLB misses with increasing contexts is greater and the TLB refill time is longer (compared to that of the L1 D-cache). For example, increasing the number of threads from 4 to 8 increases the D-TLB miss rate from 0.2% to .9% (a factor of 4.5), while the L1 D-cache miss rate rises only from 2.7% to 3.6% (a factor of 1.4). Consequently, at 8 threads, the D-TLB becomes the single largest performance-limiting hardware resource.

By combining the perfect L1 D-cache, L2 cache, and D-TLB, we show the performance impact of alleviating bottlenecks for the entire data-memory subsystem. This combination provides the best performance for both numbers of thread contexts, a 36% improvement over RR for 4 contexts and 26% for 8 contexts.

Perfect branch prediction has less impact on performances. Because multiple threads simultaneously execute on an SMT processor, each makes less progress down a mispredicted path. Consequently, the cost of the mispredictions and the gain from a perfect branch predictor is less. As the number of threads increases, the marginal benefit of perfect branch prediction declines.

Making the L1 I-cache or I-TLB perfect provided little improvement (less than 1%) in performance (data not shown). This is not surprising, given the relatively small I-cache footprints of the SPEC benchmarks.

In summary, oblivious scheduling may create bottlenecks that reduce SMT performance. Based on the results gleaned from Figure 2, we now evaluate thread-sensitive schedulers that address the contention that exists in the most bottlenecked resources: the D-TLB, the L1 data cache, the L2 cache, and the data memory subsystem as a whole. We also evaluate a scheduler that reflects total thread performance.

Program	Cache Miss-rate (%)			Wrong-path instr (% of fetched)	Instr Type (%)			IPC	
	L1-I	L1-D	L2		Int	Branch	FP	Fetch	Commit
SPEC-Int									
go	1.5	2.7	9.6	33.0	100	14	0	3.6	2.4
m88ksim	0.1	5.0	99.2	1.0	100	18	0	2.2	2.1
gcc	4.4	2.4	3.4	35.6	100	18	0	2.4	1.6
compress	0.1	3.4	2.2	57.2	80	12	20	3.3	1.4
li	0.1	1.3	0.9	21.5	100	21	0	4.1	3.2
ijpeg	0.1	0.2	65.0	4.8	100	11	0	3.9	3.7
perl	3.5	1.0	9.2	27.8	100	15	0	2.5	1.8
vortex	3.4	0.9	12.5	8.5	100	16	0	2.6	2.3
SPEC-FP									
swim	0.1	3.2	98.2	0.2	59	2	41	2.0	2.0
su2cor	0.1	6.9	29.7	28.7	66	4	34	2.8	2.0
hydro2d	0.1	9.3	61.8	0.8	67	4	33	1.7	1.7
mgrid	0.1	2.4	46.7	0.7	63	1	37	2.3	2.3
applu	0.1	3.0	54.6	1.0	48	3	52	2.2	2.2
turb3d	0.1	7.8	13.3	1.8	73	4	27	1.6	1.6
apsi	0.3	5.2	2.3	2.5	68	5	32	1.8	1.7
fpppp	5.1	0.4	0.6	2.0	54	2	46	2.8	2.7

The L2 miss-rate is relative to the total number of accesses to the L2 cache (the local miss rate). Fetch IPC refers to the rate at which instructions are fetched from the instruction stream. However, since our processor supports speculation, not all these instructions are useful. Commit IPC measures the rate at which right-path instructions are retired. The fetch IPC shows how much of the fetch bandwidth can be consumed by that program; the commit IPC is a measure of the real work done, in light of speculation.
For each benchmark, we have also shown the breakup of instructions fetched based on their type: Integer and Floating-Point. Branch refers to the subset of Integer instructions that are branch instructions.

Table 2: Characterization of SPEC95 programs

4.3 Thread-sensitive Scheduling

Before examining the thread-sensitive algorithms, it is interesting to ask whether our workload even permits an intelligent scheduler to choose jobs with different resource requirements. That is, how varied are the SPEC benchmarks in their resource demands? Table 2 sheds light on this issue by characterizing the execution time behavior of each SPEC95 program as measured on a 1-context SMT (effectively a superscalar). We can see some significant differences across the applications. For example, most integer programs (in particular *compress*, *gcc* and *go*) pose serious problems for the branch predictor (column 5), whereas most of the SPEC-FP benchmarks, being more loop-based, fetch few wrong-path instructions. This difference can also be seen in the gap between fetch and commit IPC (columns 9 and 10) for the integer programs, which indicates that they commit far fewer of their fetched instructions. The cache behavior of these programs also varies widely. For example, the L1 D-cache miss rate range from 0.2 (*ijpeg*) to

9.3 (*hydro2d*) and L2 miss rates range from 0.6 (*fpppp*) to a whopping 99.2 (*m88ksim*) percent! The effect of these variations in the individual hardware resources manifests itself in commit IPC, which also varies significantly across the programs, from 1.4 for *compress* to 3.7 for *ijpeg.* A thread-sensitive scheduler for SMT could potentially extract and exploit differences like these to alleviate contention for particular resources.

4.3.1 Thread-sensitive Schedulers

The thread-sensitive schedulers we evaluated are discussed below. In general, each algorithm focuses on a specific component metric, attempting to optimize that metric. Every quantum, the algorithm *greedily* schedules the *k* threads that have the best statistics as determined by the particular algorithm. All metrics are computed using values gathered by thread-specific hardware performance counters. The algorithms and hardware support required are listed below:

- *Miss rate in the L1 Data Cache* (G_D1MR), *L2 Cache* (G_L2MR) *and D-TLB* (G_DTLBMR)
  These three greedy algorithms choose the threads that have experienced the smallest miss rates for their respective memory component (e.g., G_L2MR chooses the threads with the lowest per-thread L2 miss rates). Maintaining per-thread statistics is feasible for the L1 caches and the D-TLB, because they are virtually addressed and need access to the thread ID in any case. However, since the L2 cache is physically addressed, it does not normally use the requesting thread ID. Therefore, G_L2MR may require the hardware to forward the thread ID to the miss counters on an L1 miss.

- *Data Memory Subsystem* (G_ADMAT)
  Given that all hardware structures for data (the L1 data cache, the L2 cache, and D-TLB) are problem areas, we also examine the average data memory access time (ADMAT), which combines the performance of all these components. The G_ADMAT algorithm picks the *k* threads with the least ADMAT values. A cheap implementation would keep two per-thread counters: one that accumulates the number of stalled memory operations each cycle and another for the total number of memory operations issued. Dividing the former by the latter gives us the average (the divide only needs to be performed once per scheduling quantum).

- *IPC* (G_IPC)
  This scheme directly uses a per-thread IPC measurement. The commit IPC achieved by a thread allows us to combine all the factors affecting a thread's performance.

  Because we have separate integer and floating-point queues and functional units, we schedule for each type of hardware separately in order to maximize the utilization of both. In other words, we compute separate integer and floating-point IPC for each thread. The scheduler then picks the threads that have the highest IPC -- half of them based on their integer IPC and half based on the FP IPC. The hardware required for this scheme consists of per-thread counters that are updated in the commit pipeline stage, one each for the integer and FP instruction counts. If the instruction type is not normally available in the commit stage, we can propagate this one bit of information from the decode stage, where it is computed.

- *Complementary Hardware Resources* (DTLBMR_MIX, ADMAT_MIX, IPC_MIX)

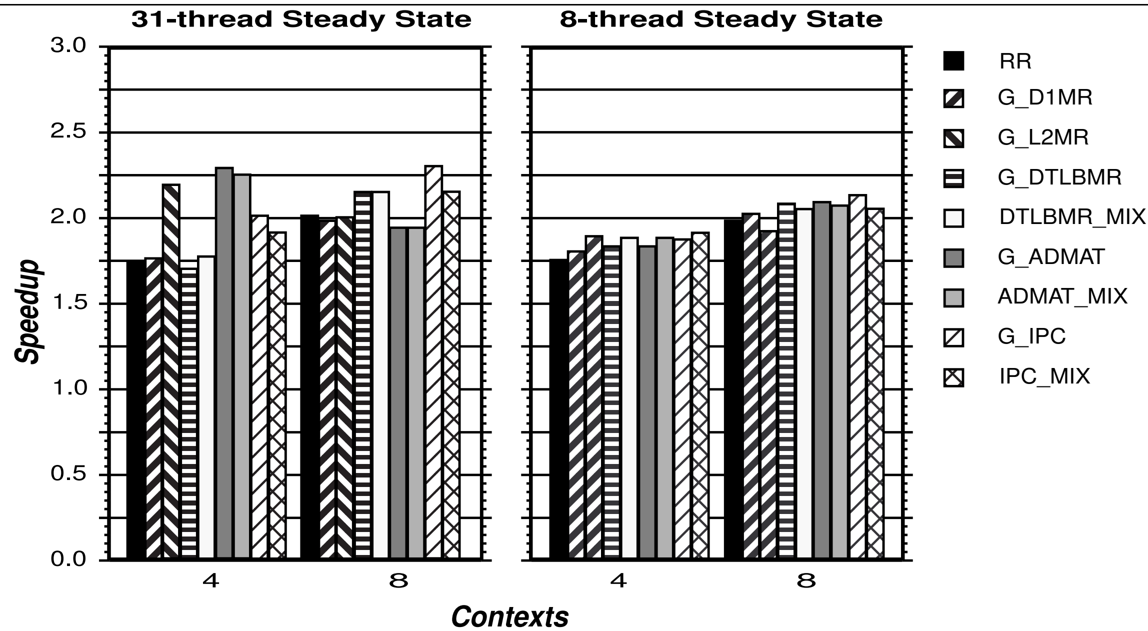

FIGURE 3. Speedup for SMTSteadyState **and** 8ThreadsLeft

These schemes attempt to utilize complementary hardware resources by choosing *k*/2 best and *k*/2 worst threads each quantum. IPC_MIX chooses threads based on total IPC, whereas its greedy counterpart, G_IPC, picked threads based on integer and floating point IPC separately.

4.3.2 Performance of Thread-sensitive Schedulers

Figure 3 shows the speedup achieved by each thread-sensitive scheduler. From the figure, we see that the schedulers achieve a wide range of performance, on average 18% across the different SMT configurations and workloads. With one exception, the schedulers that reflect total performance, G_IPC and IPC_MIX, consistently achieve the best speedups, improving on a simple RR scheme by 9% on average, with a maximum of 15%. By itself, G_IPC has an average improvement of 11% and reaches a processor utilization in the realm of 6.6 IPC. Only at 4 contexts for SMT-SteadyState does the performance of G_IPC and IPC_MIX drop below another scheduler.

The resource-specific algorithms are not as effective in achieving good IPC, and surprisingly sometimes do *worse* than RR, particularly with more thread contexts, which is exactly when we would expect them to do well. To understand this, we need to examine in more detail the performance of the resource-specific, thread-sensitive algorithms. A resource-specific algorithm's performance depends on three factors: (a) how critical its targeted resource is to overall machine utilization, (b) how well it is able to alleviate contention for that resource, and (c) how doing so affects the usage of other resources. In this section we will us focus on (b) and (c).

Table 3 shows the miss rates and branch mispredictions induced by our greedy schedulers in the shared hardware structures for the SMTSteadyState simulation with 8 contexts. (Similar results hold for 8ThreadsLeft.) First, the data indicate the extent to which resource-specific, thread-sensitive schedulers can impact the component performance metrics. For example, we see wide variation in branch prediction accuracy (1.1% to 5.5%), L2 miss rate (3.8%

Scheduler	Miss Rate (%)			Branch misprediction (%)	% wrong-path instructions	Speedup
	L1 D-cache	Data TLB	L2 cache			
RR	3.6	0.8	41.0	5.4	3.6	2.0
G_D1MR	**3.7**	0.1	22.2	4.3	2.7	2.0
G_DTLBMR	4.9	**0.1**	9.4	3.4	2.2	2.2
G_L2MR	5.3	0.2	**3.8**	4.6	5.0	2.0
G_ADMAT	5.1	0.2	4.6	5.5	4.5	2.0
G_IPC	3.8	0.1	31.3	1.1	0.5	2.3

Table 3: Comparing shared component usage across thread schedulers for SMTSteadyState, 8 contexts. **bold** = component addressed by a resource-specific thread scheduler.

to 41.0%) and D-TLB miss rate (0.1% to .8%) across the different algorithms. Second, and more important for our analysis, the resource-specific algorithms succeed in reducing contention for their specific resource, usually relative to both RR and the other resource-specific schedulers, with G_L2MR offering the most significant (relative) improvement. In spite of this, all cache-oriented algorithms (G_D1MR, G_L2MR and G_ADMAT) perform less well than G_IPC. The main reason is that greedily focusing on one resource can create or expose other resource bottlenecks, which then become dominant factors in constraining performance. For example, Table 3 shows that, although it minimizes the L2 miss rate, G_L2MR has the highest L1 data miss rate of any scheduler, almost 1.5 times that of RR. In addition, it is among the worst in branch-prediction behavior, resulting in the highest percentage of wrong-path instructions, almost 40% over that of RR. Similarly, G_D1MR forfeits its advantage in L1 data misses by shifting the bottleneck to the L2 cache.

With G_DTLBMR, on the other hand, factor (c) plays a smaller role. Table 3 shows that focusing on the D-TLB can actually reduce contention for shared resources like branch prediction and the L1 D-cache.

In line with these results, the thread-sensitive algorithms that attempt to schedule together threads with complimentary requirements for particular hardware resources (DTLBMR_MIX and ADMAT_MIX) did not usually meet expectations. They varied little from their greedy counterparts, sometimes a little better, sometimes a little worse.

Differences among the schedulers becomes less apparent in the two tail workloads, SSTail and 8Tail. G_IPC and IPC_MIX either remain the thread-sensitive schedulers of choice, or are replaced by ADMAT_MIX, the worst choice in SMTSteadyState. The consequence for executing all instructions to completion (RunToCompletion), despite its being a short run in an SMT environment, is that G_IPC still obtains the best performance.

Overall, the best algorithm is G_IPC, which achieves good performance across different configurations. G_IPC directly addresses our bottom-line performance criterion, instruction throughput. In doing so, it indirectly accounts

Scheduler	Context switches	Commit IPC	Speedup
G_D1MR	144	6.1	2.0
G_DTLBMR	115	6.7	2.2
DTLBMR_MIX	117	6.7	2.2
G_L2MR	125	6.0	2.0
G_ADMAT	147	6.3	2.0
ADMAT_MIX	147	6.3	2.0
G_IPC	151	6.6	2.3
IPC_MIX	117	6.3	2.2
RR	3008	5.8	2.0
RR-by-1	370	6.0	2.1
Random	2151	5.8	2.0

Table 4: Number of context switches for SMTSteadyState, 8 contexts. Commit IPC is shown in addition to speedup, so that the oblivious and thread-sensitive schedulers can be compared.

for many processor resources at once. It not only achieves good processor utilization, but also reduces contention for those components that matter to SMT performance (such as the D-TLB and L1 D-cache).

4.4 Cache Affinity

As previously mentioned, scheduling policies using cache affinity have been widely considered in shared-memory multiprocessors. In SMT, however, cache affinity is less of an issue, because all threads share the cache hierarchy. Table 4 provides some evidence for this; it shows the number of context switches and the IPC incurred by each algorithm when simulated on the LARGE configuration. Although fewer context switches implies less perturbation in the caches, and thus preserves temporal cache affinity, the data argues that this is a second-order effect on SMT. First, the difference in instruction throughputs between the oblivious and thread-sensitive thread schedulers is too small to be explained by the order of magnitude difference in their number of context switches. In addition, several specific cases serve as counter examples. RR has over 8 times the number of context switches as RR-by-1, yet their performance is nearly identical (see also Figure 1). G_D1MR (plus G_ADMAT and ADMAT_MIX) and G_IPC are relatively close in their number of context switches, but G_IPC has a 15% higher speedup. The conclusion we draw is that, although temporal cache affinity does help improve performance, selecting threads based on their characteristics is more important.

4.5 Summary

In this section we examined both oblivious and thread-sensitive schedulers. Our results show that thread-sensitive scheduling can have an advantage on SMT processors, boosting instruction throughput by up to 15% over round-robin in the best cases of G_IPC and IPC_MIX. Other hardware-resource-specific algorithms perform less well. Although they reduce inter-thread contention for their particular resource, in doing so, they create contention for other resources. Consequently, their performance benefit is lower, and they may even degrade performance relative to the oblivious schemes.

5 Fairness

Most of the thread-sensitive algorithms we defined greedily pick the best threads according to their corresponding performance metrics. However, as greedy algorithms, they can cause thread starvation. We would therefore like to measure how much starvation our schedulers can cause, and prevent it if possible. For a particular thread, starvation is defined relative to a baseline that represents how many quanta that thread *should* have been scheduled if the scheduler were fair. In order to derive a measure of this *Ideal* metric, we introduce the following terminology:

Set of threads in the workload :	$T = \{t_1, t_2, ..., t_n\}$
Set of time steps in the schedule :	$Q = \{1, 2, .., q_{max}\}$
Set of time quanta in which thread t_i was runnable :	$Q_i \subseteq Q$
Number of contexts :	k
Number of runnable threads in quantum j :	n_j

In an ideal fair schedule, each of the n_j runnable threads in a particular quantum j should receive k/n_j of that quantum. Without redefining the definition of a scheduler quantum, it is not possible to do this fractional allocation in a given quantum. However, we can consider the total allocation over several quanta and aim to achieve that (in the limit). Thus, $Ideal_i$, the number of quanta that thread t_i *should* have been scheduled, can be computed as:

$$Ideal_i = \sum_{j \in Q_i} \frac{k}{n_j}.$$

Let $Actual_i$ be the actual number of quanta that a thread is scheduled. The quantity $d_i = \left|1 - \frac{Actual_i}{Ideal_i}\right|$ is the separation of the ratio $\frac{Actual_i}{Ideal_i}$ from the ideal ratio of 1. We can judge the fairness of the schedule by measuring the mean of these ratios across the threads: $F_{mean} = \frac{1}{n}\sum_{1}^{n} d_i$

In a fair schedule, F_{mean} should be close to 0. F_{mean} indicates how far a thread is from the ideal, on average. Note that maintaining the values of $Actual_i$ and $Ideal_i$ is not computationally intensive, each of them requiring a simple increment to counters in the per-thread information maintained by the OS.

In Figure 4 we plot F_{mean} for our greedy schedulers. As we suspected, they can be quite unfair, with a thread being 85% to 130% away from its ideal allocation of quanta. The fairness of the schedulers improves when there are more hardware contexts available. With few contexts, it is easy to starve many threads for long periods. As the number of

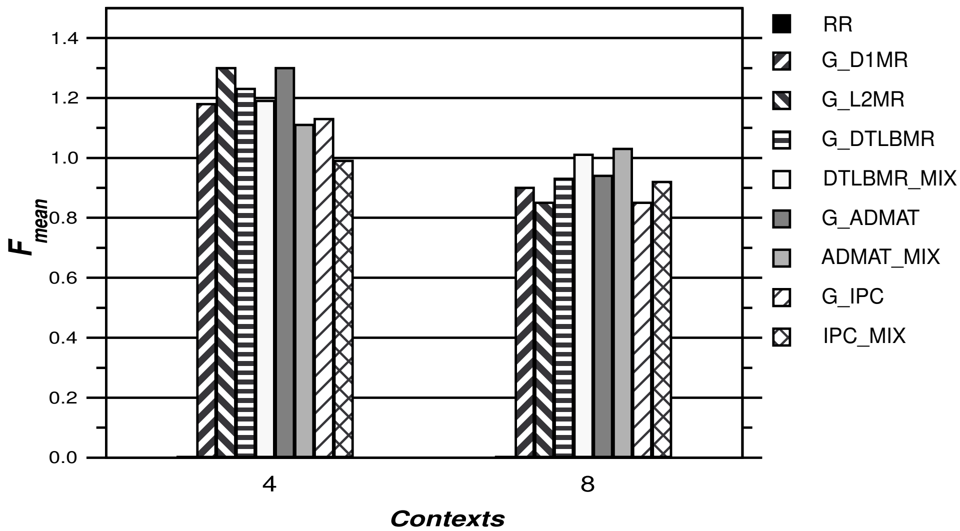

FIGURE 4. Fairness measurement of thread-sensitive schedulers for 8ThreadsLeft. (Note: RR is 0.**)**

contexts increases, more slots become available to the scheduler, and the possibility that a particular thread will be starved is reduced.

We can mitigate the amount of starvation induced by these algorithms by forcing starved threads to be scheduled even if they do not meet the greedy criteria. To do this, we modified the scheduler as follows: if there were any threads such that $\frac{Actual_i}{Ideal_i} \leq \alpha$, ($0 \leq \alpha \leq 1$), we first scheduled those threads. A particular greedy heuristic is then used to schedule any leftover contexts.

This scheme forces $\frac{Actual_i}{Ideal_i} \geq \alpha$ for each thread t_i.[1] The larger α is, the less deviation is permitted from an ideal "fair" schedule. Figure 5 shows the effects of adding this anti-starvation metric to G_IPC with $\alpha \in \{0.3, 0.6, 0.9\}$. In all cases, the IPC drops off systematically with a stronger anti-starvation measure. In particular, the strongest α that we used reduces the performance of G_IPC close to that of RR. The slight dip of G_IPC relative to RR at α=0.9 is not significant; we have observed differences in speedup of that magnitude even across runs of the same scheduler with slightly different initial configurations.

Thus we see that there is a very clear trade-off between guaranteeing some fairness in the OS scheduler and in improving CPU utilization. By using G_IPC and adjusting the value of α, a system administrator can adjust the relative preference between IPC and fairness to achieve the best balance. For a throughput-oriented machine, we can prevent total starvation by using a low value of α, with only a small degradation in performance. For instance, in Figure 5, α = 0.3 results in a degradation of less than 2% relative to G_IPC for the configurations shown. However, a policy favoring fairness would be best off with a simple RR scheme. Using a high value of α does not provide significant

1. Note that we do not explicitly prohibit favoring certain threads, although such measures could be added as well.

speedup benefits over RR.

6 Summary

Simultaneous multithreading is unique in the level of fine-grained inter-thread resource sharing it permits, because instructions from multiple threads execute in a single cycle. SMT threads therefore have a higher potential for resource interference than do threads on a CMP, and hence thread scheduling for SMTs is an important operating system activity.

We propose that the SMT OS scheduler take advantage of hardware counters that are commonly available on modern CPUs when deciding which threads to schedule. In this paper we investigated the performance of some simple, greedy algorithms that utilize this performance feedback information. For the configurations and workloads we measured, our results showed that greedily selecting the highest-throughput threads improved CPU utilization over round-robin by 7% to 15% (11% on average). Since its implementation cost is minimal, both in hardware (2 counters per thread) and thread scheduling software (sampling the counters), it seems a trade-off worth making.

We have also shown that temporal cache affinity scheduling is not a reliable method for obtaining performance improvements on an SMT. The G_IPC algorithm is able to consistently achieve good IPC in spite of having more context switches and worse L2 cache behavior than other, worse-performing schedulers. Thus, scheduling for cache affinity (another common SMP issue) has little payoff for an SMT processor.

The cost of using a greedy algorithm is that certain threads may be starved. The thread-sensitive algorithms studied here can be easily modified to prevent this, albeit at some cost in performance. Our results show that for a throughput-oriented machine, we can provide weak fairness guarantees while barely degrading performance.

Bibliography

[1] A. Agarwal. Performance tradeoffs in multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5), September 1992.

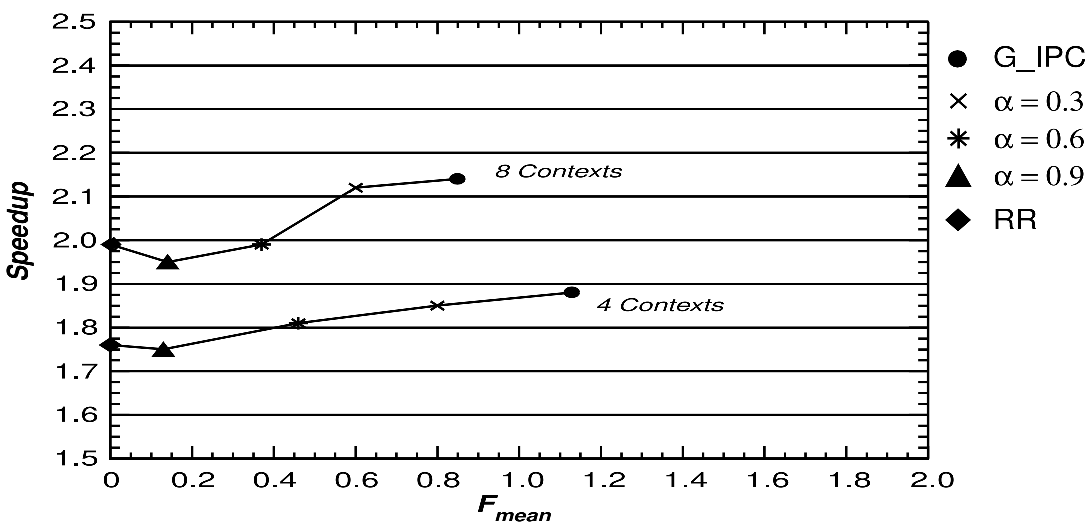

FIGURE 5. The trade-off between IPC and Fairness for G_IPC (8ThreadsLeft).

[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] G. Alverson, R. Alverson, D. Callahan, B. Koblenz, A. Porterfield, and B. Smith. Exploiting heterogeneous parallelism on a multithreaded multiprocessor. In *Supercomputing '91 Workshop on Multithreaded Architectures*, November 1991.

[4] M. Bach, editor. *The Design of the Unix Operating System*. Prentice-Hall, 1996.

[5] D. Black. Scheduling support for concurrency and parallelism in the Mach operating system. *IEEE Computer*, 23(5), May 1990.

[6] B. Boothe and A. Ranade. Improved multithreading techniques for hiding communication latency in multiprocessors. In *19th Annual International Symposium on Computer Architecture*, 1992.

[7] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum. Scheduling and page migration for multiprocessor compute servers. In *6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

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

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

[10] 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.

[11] J. Fiske. Thread scheduling mechanisms for multiple-context parallel processors. Technical report, Ph.D. thesis, M.I.T., June 1995.

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

[13] 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.

[14] 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.

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

[16] M. McKusick, K. Bostic, M. Karels, and J. Quarterman, editors. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley Publishing, 1997.

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

[18] R. Thekkath and S. Eggers. The effectiveness of multiple hardware contexts. In *6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[19] J. Torrelas, A. Tucker, and A. Gupta. Benefits of cache-affinity scheduling in shared-memory multiprocessors: A summary. In *1993 ACM Sigmetrics*, May 1993.

[20] A. Tucker and A. Gupta. Process control and scheduling issues for multiprogrammed shared memory multiprocessors. In *Symposium on Operating Systems Principals*, December 1989.

[21] 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.

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

[23] R. Vaswani and J. Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Symposium on Operating Systems Principals*, October 1991.