An Evaluation of Speculative Instruction Execution on Simultaneous Multithreaded Processors

STEVEN SWANSON, LUKE K. McDOWELL, MICHAEL M. SWIFT,
SUSAN J. EGGERS and HENRY M. LEVY
University of Washington

Modern superscalar processors rely heavily on speculative execution for performance. For example, our measurements show that on a 6-issue superscalar, 93% of committed instructions for SPECINT95 are speculative. Without speculation, processor resources on such machines would be largely idle. In contrast to superscalars, simultaneous multithreaded (SMT) processors achieve high resource utilization by issuing instructions from multiple threads every cycle. An SMT processor thus has two means of hiding latency: speculation and multithreaded execution. However, these two techniques may conflict; on an SMT processor, wrong-path speculative instructions from one thread may compete with and displace useful instructions from another thread. For this reason, it is important to understand the trade-offs between these two latency-hiding techniques, and to ask whether multithreaded processors should speculate differently than conventional superscalars.

This paper evaluates the behavior of instruction speculation on SMT processors using both multiprogrammed (SPECINT and SPECFP) and multithreaded (the Apache Web server) workloads. We measure and analyze the impact of speculation and demonstrate how speculation on an 8-context SMT differs from superscalar speculation. We also examine the effect of speculation-aware fetch and branch prediction policies in the processor. Our results quantify the extent to which (1) speculation is critical to performance on a multithreaded processor because it ensures an ample supply of parallelism to feed the functional units, and (2) SMT actually enhances the effectiveness of speculative execution, compared to a superscalar processor by reducing the impact of branch misprediction. Finally, we quantify the impact of both hardware configuration and workload characteristics on speculation's usefulness and demonstrate that, in nearly all cases, speculation is beneficial to SMT performance.

Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures; C.4 [**Performance of Systems**]; C.5 [**Computer System Implementation**]

General Terms: Design, Measurement, Performance

Additional Key Words and Phrases: Instruction-level parallelism, multiprocessors, multithreading, simultaneous multithreading, speculation, thread-level parallelism

---

1. INTRODUCTION

Instruction speculation is a crucial component of modern superscalar processors. Speculation hides branch latencies and thereby boosts performance by executing the likely branch path without stalling. Branch predictors, which provide accuracies up to 96% (excluding OS code) [Gwennap 1995], are the key to effective speculation. The primary disadvantage of speculation is that some processor resources are invariably allocated to useless, wrong-path instructions that must be flushed from the pipeline. However, since resources on superscalars are often underutilized because of low single-thread instruction-level parallelism (ILP) [Tullsen et al. 1995; Cvetanovic and Kessler 2000], the benefit of speculation far outweighs this disadvantage and the decision to speculate as aggressively as possible is an easy one.

In contrast to superscalars, simultaneous multithreading (SMT) processors [Tullsen et al. 1995, 1996] operate with high processor utilization, because they issue and execute instructions from multiple threads each cycle, with all threads dynamically sharing hardware resources. If some threads have low ILP, utilization is improved by executing instructions from additional threads; if only one or a few threads are executing, then all critical hardware resources are available to them. Consequently, instruction throughput on a fully loaded SMT processor is two to four times higher than on a superscalar with comparable hardware on a variety of integer, scientific, database, and web service workloads [Lo et al. 1997a,b; Redstone et al. 2000].

With its high hardware utilization, speculation on an SMT may harm rather than improve performance. This would be particularly true for SMT's likely-targeted application domain: highly threaded, high-performance servers, with all hardware contexts occupied. In this scenario, speculative (and potentially wasteful) instructions from one thread may compete with useful, non-speculative instructions from other threads for highly utilized hardware resources, and in some cases displace them, lowering performance. This raises the possibility that SMT might be able to capitalize on its inherent latency-hiding abilities to reduce the need for speculation. If SMT could do without speculation while maintaining the same level of performance, it might dispense with the complicated control necessary to recover from mispeculations. To resolve this issue, it is important to understand the behavior of speculation on an SMT processor and the extent to which it helps or hinders performance.

In investigating speculation on SMT, this paper makes three principle contributions:

—A careful analysis of the interactions between speculation and multithreading.
—A detailed simulation study of a wide range of alternative, speculation-aware SMT fetch policies.
—A characterization of the conditions (both hardware configuration and workloads) under which speculation is helpful to SMT performance.

Our analyses are based on five different workloads (including all operating system code): SPECINT95, SPECFP95, a combination of the two, the Apache Web server, and a synthetic workload that allows us to manipulate basic-block length and available ILP. Using these workloads, we carefully examine how speculative instructions behave on SMT, as well as how and when SMT should speculate.

We attempt to improve speculation performance on SMT by reducing wrong-path speculative instructions, either by not speculating at all or by using speculation-aware fetch policies (including policies that incorporate confidence estimators). To explain the results, we investigate which hardware structures and pipeline stages are affected by speculation, and how speculation on SMT processors differs from speculation on a traditional superscalar. Finally, we explore the boundaries of speculation's usefulness on SMT by varying the number of hardware threads, the number of functional units, and the cache capacities, and by using synthetic workloads to change the branch frequency and ILP within threads.

After describing the methodology for our experiments in the next section, we present the basic speculation results and explain why and how speculation benefits SMT performance; this section also discusses alternative fetch and prediction schemes and shows why they fall short. Section 4 continues our analysis of speculation, exploring the effects of software and microarchitectural parameters on speculation. Finally, Section 5 discusses related work and Section 6 summarizes our findings.

2. METHODOLOGY

2.1 Simulator

Our SMT simulator is based on the SMTSIM simulator [Tullsen 1996] and has been ported to the SimOS framework [Rosenblum et al. 1995; Redstone et al. 2000; Compaq 1998]. It simulates the full pipeline and memory hierarchy, including bank conflicts and bus contention, for both the applications and the operating system.

The baseline configuration for our experiments is shown in Table I. For most experiments we used the ICOUNT fetch policy [Tullsen et al. 1996]. ICOUNT gives priority to threads with the fewest number of instructions in the pre-issue stages of the pipeline and fetches 8 instructions (or to the end of the cache line) from each of the two highest priority threads. From these instructions, it chooses to issue up to 8, selecting from the highest priority thread until a branch instruction is encountered, then taking the remainder from the second thread. In addition to ICOUNT, we also experimented with three alternative fetch policies. The first does not speculate at all, that is, instruction fetching for a particular thread stalls until the branch is resolved; instead, instructions are selected only from the non-speculative threads using ICOUNT. The second favors non-speculating threads by fetching instructions from threads whose next instructions are non-speculative before fetching from threads

Table I. SMT Parameters

CPU	
Thread Contexts	8
Pipeline	9 stages, 7 cycle misprediction penalty.
Fetch Policy	8 instructions per cycle from up to 2 contexts (the ICOUNT scheme of Tullsen et al. [1996])
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
Branch Predictor	McFarling-style, hybrid predictor [McFarling 1993] (shared among all contexts)
Local Predictor	4K-entry prediction table, indexed by 2K-entry history table
Global Predictor	8K entries, 8K-entry selection table
Branch Target Buffer	256 entries, 4-way set associative (shared among all contexts)
Cache Hierarchy	
Cache Line Size	64 bytes
Icache	128KB, 2-way set associative, dual-ported, 2 cycle latency
Dcache	128KB, 2-way set associative, dual-ported (from CPU, r&w), single-ported (from the L2), 2 cycle latency
L2 cache	16MB, direct mapped, 23 cycle latency, fully pipelined (1 access per cycle)
MSHR	32 entries for the L1 cache, 32 entries for the L2 cache
Store Buffer	32 entries
ITLB & DTLB	128-entries, fully associative
L1-L2 bus	256 bits wide
Memory bus	128 bits wide
Physical Memory	128MB, 90 cycle latency, fully pipelined

with speculative instructions. The third uses branch confidence estimators to favor threads with high-confidence branches. In all cases, ICOUNT breaks ties.

Our baseline experiments used the McFarling branch prediction algorithm [McFarling 1993] used on modern processors from Hewlett Packard; for some studies we augmented this with confidence estimators. Our simulator speculates past an unlimited number of branches, although in practice it speculates only past 1.4 on average and almost never (less than 0.06% of cycles) past more than 5 branches.

In exploring the limits of speculation's effectiveness, we also varied the number of hardware contexts from 1 to 16. Finally, for the comparisons between SMT and superscalar processors we use a superscalar with the same hardware

components as our SMT model but with a shorter pipeline, made possible by the superscalar's smaller register file.

2.2 Workload

We use three multiprogrammed workloads: SPECINT95, SPECFP95 [Reilly 1995], and a combination of four applications from each suite, INT+FP. In addition we used the Apache web server (version 1.3), an open source web server run by the majority of web sites [Hu et al. 1999]. We drive Apache with SPECWEB96 [System Performance Evaluation Cooperative 1996], a standard web server performance benchmark, configured with two client machines each running 64 client processes. Each workload serves a different purpose in the experiments. The integer benchmarks are our dominant workload and were chosen because their frequent, less predictable branches (relative to floating point programs) provide many opportunities for speculation to affect performance. Apache was chosen because over three-quarters of its execution occurs in the operating system, whose branch behavior is also less predictable [Agarwal et al. 1988; Gloy et al. 1996], and because it represents the server workloads that constitute one of SMT's target domains. We selected the floating point suite because it contains loop-based code with large basic blocks and more predictable branches than integer code, providing an important perspective on workloads where speculation is more beneficial. Finally, following the example of Snavely and Tullsen [2000], we combined floating point and integer code to understand how interactions between different types of applications affect our results.

We also used a synthetic workload to explore how branch prediction accuracy, branch frequency, and the amount of ILP affect speculation on an SMT. The synthetic program executes a continuous stream of instructions separated by branches. We varied the average number and independence of instructions between branches, and the prediction accuracy of the branches is set by a command line argument to the simulator.

We execute all of our workloads under the Compaq Tru64 Unix 4.0d operating system; the simulation includes all OS privileged code, interrupts, drivers, and Alpha PALcode. The operating system execution accounts for only a small portion of the cycles executed for the SPEC workloads (about 5%), while the majority of cycles (77%) for the Apache Web server are spent inside the OS managing the network and disk.

Most experiments include 200 million cycles of simulation starting from a point 600 million instructions into each program (simulated in 'fast mode'). The synthetic benchmarks, owing to their simple behavior and small size (there is no need to warm the L2 cache), were simulated for only 1 million cycles each. Other researchers have demonstrated that, for SPECINT95 and Apache, our segments are well past the beginning of steady-state execution [Redstone et al. 2000]. To ensure that the portions of execution for the other benchmarks are representative, we performed some longer simulations and found they had no significant effect on our results. For machine configurations with more than 8 contexts, we ran multiple instances of some of the applications.

2.3 Metrics and Fairness

Changing the fetch policy of an SMT necessarily changes which and in what order instructions execute. Different policies affect each thread differently and, as a result, they may execute more or fewer instructions over a 200 million cycle simulation. Consequently, directly comparing the total IPC with two different fetch policies may not be fair, since a different mix of instructions is executed, and the contribution of each thread to the bottom-line IPC changes.

We resolved this problem by following the example set by the SPECrate metric [System Performance Evaluation Cooperative 2000] and averaging performance across threads instead of cycles. The SPECrate is the percent increase in throughput (IPC) relative to a baseline for each thread, combined using the geometric mean. Following this example, we computed the geometric mean of the threads' speedups in IPC relative to their performance on a machine using the baseline ICOUNT fetch policy and executing the same threads on the same number of contexts. Finally, because our workload contains some threads (such as interrupt handlers) that run for only a small fraction of total simulation cycles, we weighted the per-thread speedups by the number of cycles the thread was scheduled in a context.

Using this technique we computed an average speedup across all threads. We then compared this value to a speedup calculated just using the total IPC of the workload. We found that the two metrics produced very similar results, differing on average by just 1% and at most by 5%. Moreover, none of the performance trends or conclusions changed based on which metric was used. Consequently, for the configurations we consider, using total IPC to compare performance is accurate. Since IPC is a more intuitive metric to discuss than the speedup averaged over threads, in this paper we report only the IPC for each experiment.

3. SPECULATION ON SMT

This section presents the results of our simulation experiments on instruction speculation for SMT. Our goal is to understand the trade-offs between two alternative means of hiding branch delays: instruction speculation and SMT's ability to execute instructions from multiple threads each cycle. First, we compare the performance of an SMT processor with and without speculation and analyze the differences between these two options. Then we discuss the impact of speculation-aware fetch policies and the use of branch prediction confidence estimators on speculation performance.

3.1 The Behavior of Speculative Instructions

As a first task, we modified our SMT simulator to turn off speculation (i.e., the processor never fetches past a branch until it has resolved it) and compared the throughput in instructions per cycle on our four workloads with a speculative SMT CPU. The results of these measurements, seen in Table II, show that speculation benefits SMT performance on all four workloads—the speculative SMT achieves performance gains of between 9% and 32% over the non-speculative processor. Apache, with its small basic blocks and poor branch

Table II. Effect of Speculation on SMT. We Simulated Each of the Four Workloads on
Machines with and Without Speculation

	SPECINT95	SPECFP95	INT+FP	Apache
IPC with speculation	5.2	6.0	6.0	4.5
IPC without speculation	4.2	5.5	5.5	3.4
Improvement from speculation	24%	9%	9%	32%

prediction, derives the most performance from speculation, while the more predictable floating benchmarks benefit least. SMT's benefit from speculation is far lower than the 3-fold increase in performance that superscalars derive from speculation, but it falls on the same side of the trade-off between the increased ILP that speculation provides and the resources it wastes.

Speculation can have different effects throughout the pipeline and the memory system. For example, speculation could pollute the cache with instructions that will never be executed or, alternatively, prefetch instructions before they are needed, eliminating future cache misses. None of these effects appear in our simulations, and turning off speculation never altered the percentage of cache hits by more that 0.4%.

To understand how speculative instructions execute on an SMT processor and how they benefit its performance and resource utilization, we categorized instructions according to their speculation behavior:

—non-speculative instructions are those fetched non-speculatively—they always perform useful work;

—correct-path-speculative instructions are fetched speculatively, are on the correct path of execution, and therefore accomplish useful work;

—wrong-path-speculative instructions are fetched speculatively, but lie on incorrect execution paths, are thus ultimately flushed from the execution pipeline and consequently waste hardware resources.

Using this categorization, we followed all instructions through the execution pipeline. At each pipeline stage we measured the average number of each of the three instruction types that leaves that stage each cycle. We call these values the correct-path-speculative, wrong-path-speculative, and non-speculative per-stage IPCs. The overall machine IPC is the sum of the correct-path-speculative and non-speculative commit IPCs.

Figures 1–4 depict these per-stage instruction categories for all four workloads. While bottom line IPC of the four workloads varies considerably, the trends we describe in the next few paragraphs are remarkably consistent across all of them. For instance, although the distribution of instructions between the three categories changes, in all cases between 82 and 86% of wrong-path instructions leave the pipeline before they reach the functional units and no more than 2% of instruction executed are on the wrong path. The similarity implies that the conclusions for SPECINT95 are applicable to the other three workloads, suggesting that the behavior is fundamental to SMT, rather than being workload dependent. Because of this, we present data primarily for SPECINT95, and discuss the other workloads only when it contributes to the

(A)

(B)

Fig. 1.   Per-pipeline-stage IPC for SPECINT95, divided between correct-path-, wrong-path-, and non-speculative instructions. On top, (a) SMT with ICOUNT; on the bottom, (b) SMT with a fetch policy that favors non-speculative instructions.

(A)

(B)
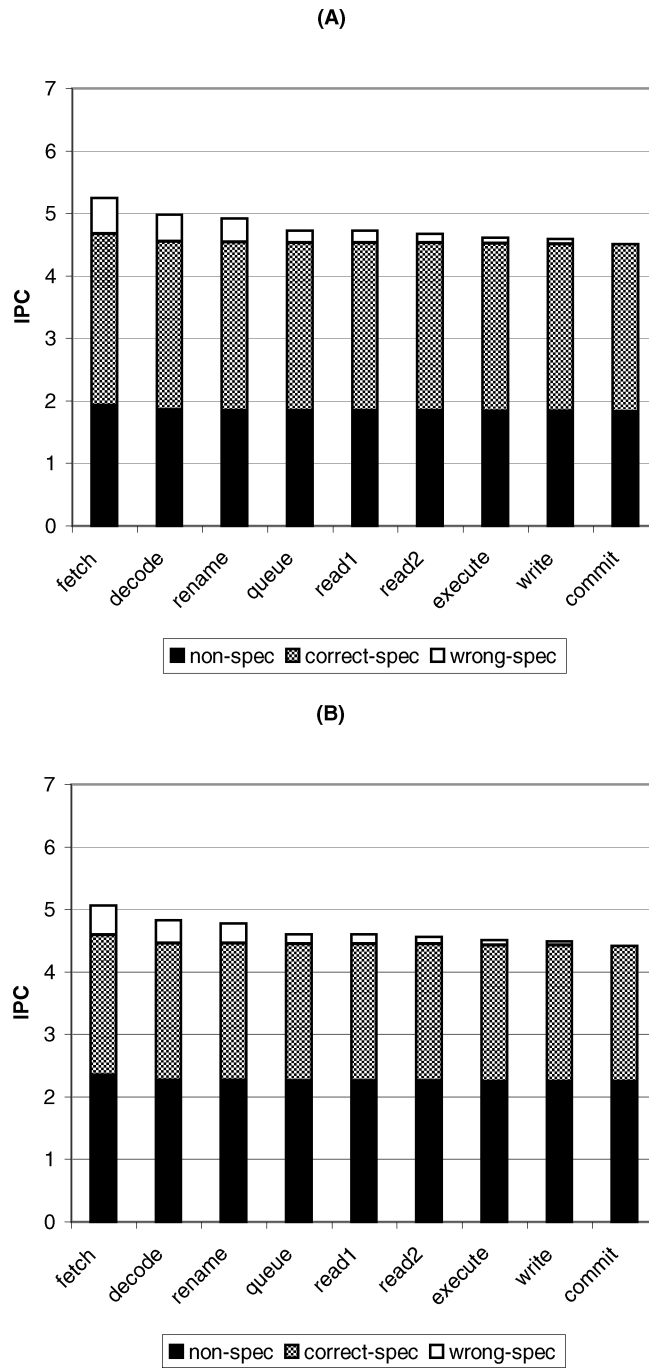
Fig. 2.   Per-pipeline-stage IPC for Apache, divided between correct-path-, wrong-path-, and non-speculative instructions. On top, (a) SMT with ICOUNT; on the bottom, (b) SMT with a fetch policy that favors non-speculative instructions.

(A)

(B)

Fig. 3.   Per-pipeline-stage IPC for SPECFP95, divided between correct-path-, wrong-path-, and non-speculative instructions. On top, (a) SMT with ICOUNT; on the bottom, (b) SMT with a fetch policy that favors non-speculative instructions.

(A)

(B)

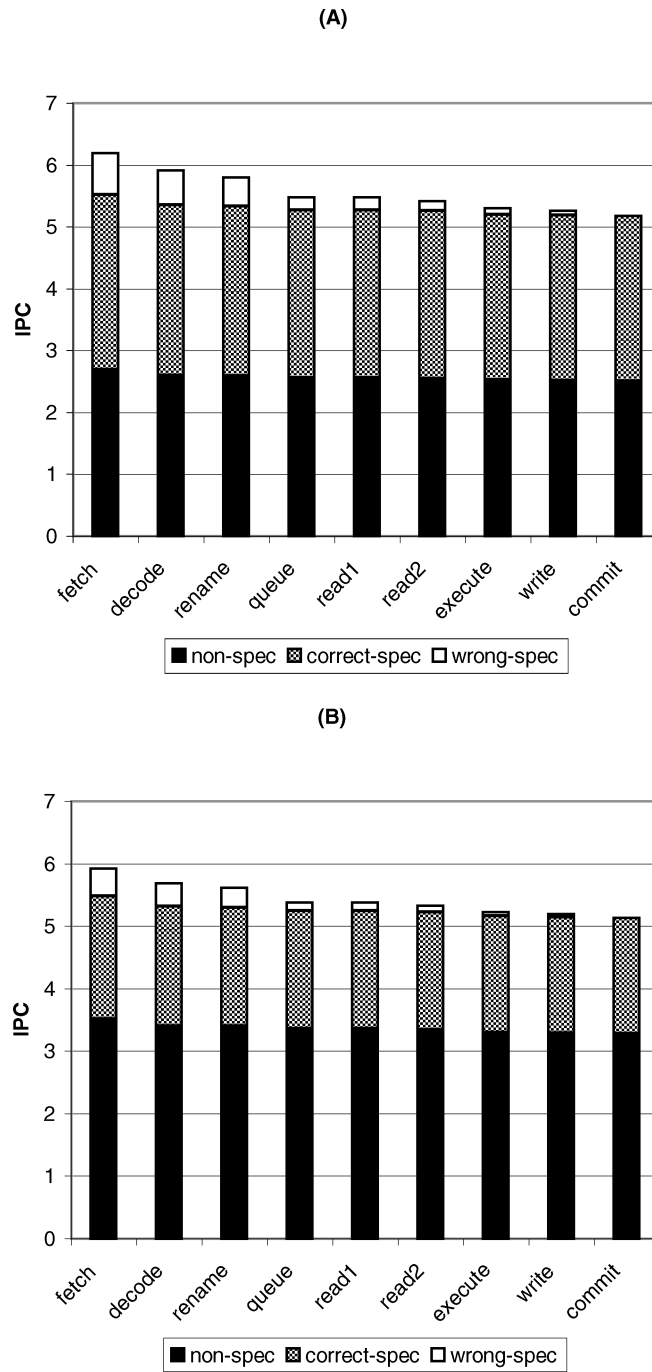
Fig. 4.   Per-pipeline-stage IPC for INT+FP, divided between correct-path-, wrong-path-, and non-speculative instructions. On top, (a) SMT with ICOUNT; on the bottom, (b) SMT with a fetch policy that favors non-speculative instructions.
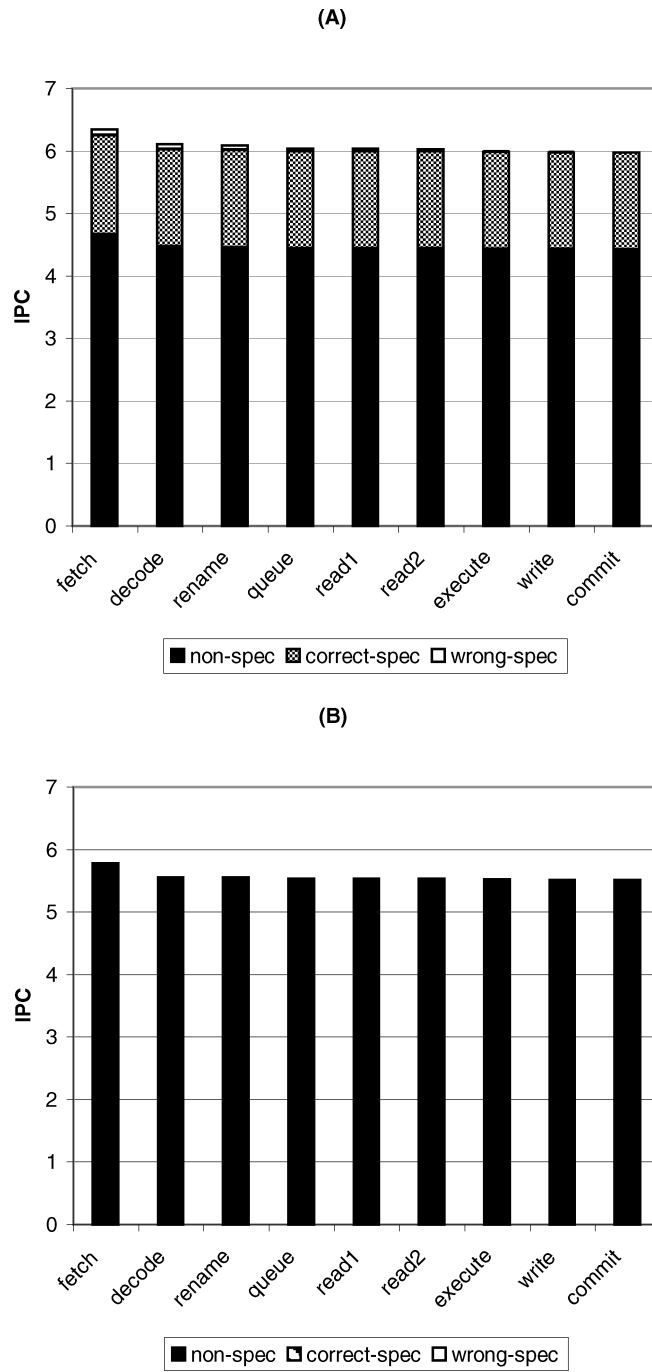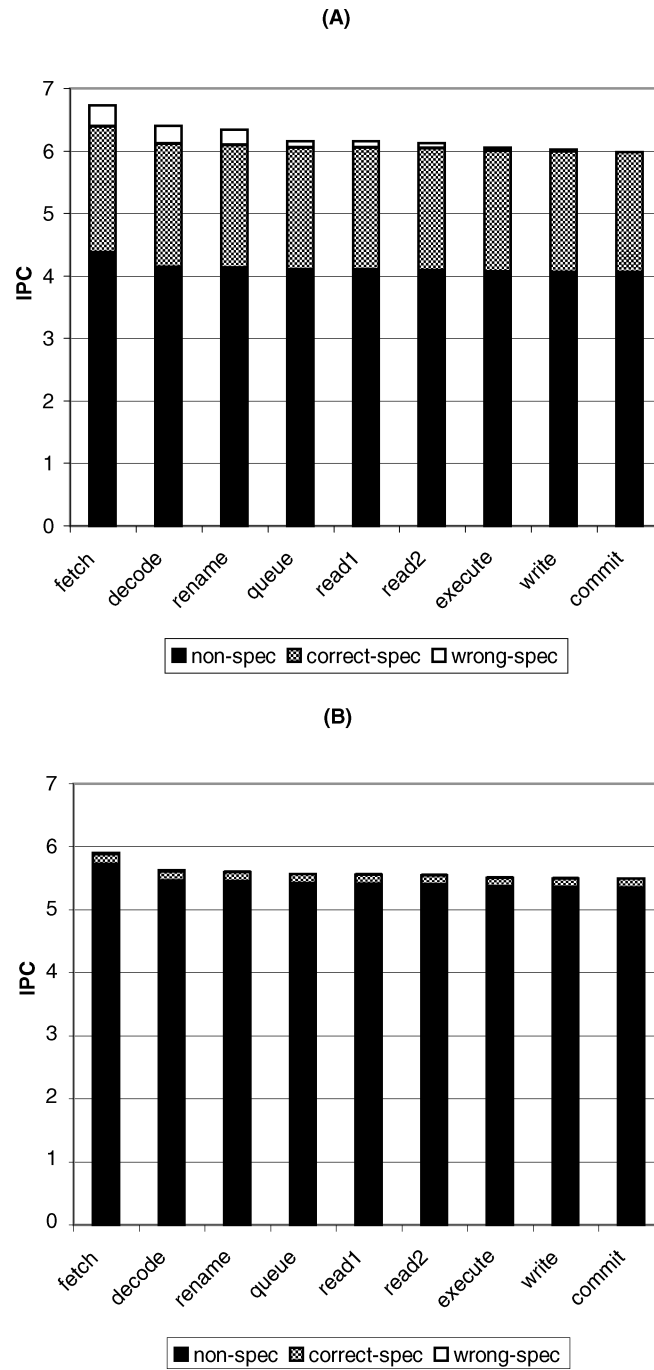
analysis. Tables VII–X in the Appendix contain a summary of the data for all fetch policies we investigated.

The upper portions of Figures 1–4 (labeled 'A') show why speculation is crucial to high instruction throughput and explain why misspeculation does not waste hardware resources. Speculative instructions on an SMT comprise the majority of instructions fetched, executed, and committed. In the case of SPECINT95 (Figure 1), for example, 57% of fetch IPC, 53% of instructions issued to the functional units, and 52% of commit IPC are speculative. (Comparable numbers for the superscalar are between 90 and 93%.) SPECFP95 and INT+FP fetch fewer speculative instructions, but they still account for a substantial portion of the instruction stream. Apache speculates the most: 63% of fetched instructions and 60% of executed instructions are speculative. Given the magnitude of these numbers and the accuracy of today's branch prediction hardware, it is not surprising that stalling until branches resolve failed to improved performance.

Speculation is particulary effective on SMT for two reasons, as SPECINT95 illustrates. First, since SMT fetches from each thread only once every 5.4 cycles on average for this workload (as opposed to almost every cycle for the single-threaded superscalar), it speculates less aggressively past branches (past 1.4 branches on average compared to 3.5 branches on a superscalar). This causes the percentage of speculative instructions fetched to decline from 93% on a superscalar to 57% on SMT. More important, it also reduces the percentage of speculative instructions on the wrong path; because an SMT processor makes less progress down speculative paths, it avoids multiple levels of speculative branches, which impose higher (compounded) misprediction rates. For the SPECINT benchmarks, for example, 19% of speculative instructions on SMT are wrong path, compared to 28% on a superscalar. Therefore, SMT receives significant benefit from speculation at a lower cost, compared to a superscalar.

Second, the data show that speculation is not particularly wasteful on SMT. Branch prediction accuracy for SPECINT95 is 88%,[1] and only 11% of fetched instructions were flushed from the pipeline. Eighty-three percent of these wrong-path-speculative instructions were removed from the pipeline before they reached the functional units, only consuming resources in the form of integer instruction queue entries, renaming registers, and fetch bandwidth. Both the instruction queue (IQ) and the pool of renaming registers are adequately sized: the IQ is only full 4.3% of cycles and renaming registers are exhausted only 0.3% of cycles. (Doubling the integer IQ for SPECINT95 reduced queue overflow to 0.4% of cycles, but raised IPC by only 1.8%, confirming that the integer IQ is not a serious bottleneck. Tullsen et al. [1996] report a similar result.) Thus, IQ entries and renaming registers are not highly contended. This leaves fetch bandwidth as the only resource that speculation wastes significantly and suggests that modifying the fetch policy might improve performance. We address this question in the next section.

Without speculation, only nonspeculative instructions use processor resources and SMT devotes no processor resources to wrong-path instructions.

---

[1]The prediction rate is lower than the value found in Gwennap [1995] because we include operating system code.

However, in avoiding wrong-path instructions, SMT leaves many of its hardware resources idle. For example, fetch stall cycles—cycles when no thread was fetched, rose almost three-fold for Apache; consequently, its per-stage IPCs dropped between 13% and 35%. Functional utilization dropped by 16% and commit IPC, the bottom-line metric for SMT performance, was 3.9, a 32% loss compared to an SMT that speculates. Our results for the other benchmarks show the same phenomena, although the other workloads benefit less from speculation. In summary, not speculating wastes more resources than mispeculating.

3.1.1 *Fetch Policies.*   It is possible that more speculation-aware fetch policies might outperform SMT's default fetch algorithm, ICOUNT, reducing the number of wrong-path instructions while increasing the number of correct-path and nonspeculative instructions. To investigate these possibilities, we compared SMT with ICOUNT to an SMT with two alternative fetch policies: one that favors nonspeculating threads and a family of fetch policies that incorporate branch prediction confidence.

3.1.2 *Favoring Nonspeculative Contexts.*   A fetch policy that favors nonspeculative contexts (see Figures 1–4) increased the proportion of nonspeculative instructions fetched by an average of 44% and decreased correct-path- and wrong-path-speculative instructions by an average of 33% and 39%, respectively. Despite the moderate shift to useful instructions (wrong-path-speculative instructions were reduced from 11% to 7% of the workload), the effect on commit IPC was negligible. This lack of improvement in IPC will be addressed again and explained in Section 3.2.

3.1.3 *Using Confidence Estimators.*   Researchers have proposed several hardware structures that assign confidence levels to branch predictions, with the goal of reducing the number of wrong-path speculations [Jacobson et al. 1996; Grunwald et al. 1998]. Each dynamic branch receives a confidence rating, a high value for branches that are usually predicted correctly and a low value for misbehaving branches. Several groups have suggested using confidence estimators on SMT to reduce wrong-path-speculative instructions and thus improve performance [Jacobson et al. 1996; Manne et al. 1998]. In our study we examined three different confidence estimators discussed in Grunwald et al. [1998]; Jacobson et al. [1996]:

—The JRS estimator uses a table that is indexed by the PC xor-ed with the global branch history register. The table contains counters that are incremented when the predictor is correct and reset on an incorrect prediction.
—The strong-count estimator uses the counters in the local and global predictors to assign confidence. The confidence value is the number of counters for the branch (0, 1, or 2) that are in a strongly-taken or strongly-not-taken state (this subsumes the both-strong and either-strong estimators in Grunwald et al. [1998]).
—The distance estimator takes advantage of the fact that mispredictions are clustered. The confidence value for a branch is the number of correct predictions that a context has made in a row (globally, not just for this branch).

Table III. Hard Confidence Performance for SPECINT95. Branch Prediction Accuracy was 88%

Confidence Estimator	Wrong-path Predictions Avoided (true negatives)	Correct Predictions Lost (false negatives)	IPC
	% of branch instructions		
No confidence estimation	0	0	5.2
JRS (threshold = 1)	2.0	6.0	5.2
JRS (threshold = 15)	7.7	38.3	4.8
Strong (threshold = 1: either)	0.7	3.9	5.1
Strong (threshold = 2: both)	5.6	31.9	4.8
Distance (threshold = 1)	1.5	6.6	5.2
Distance (threshold = 3)	3.8	16.2	5.1
Distance (threshold = 7)	5.8	27.9	4.9

There are at least two different ways to use such confidence information. In the first, hard confidence, the processor stalls a thread on a low confidence branch, fetching from other threads until the branch is resolved. In the second, soft confidence, the processor assigns a fetch priority according to the confidence of a thread's most recent branch.

Hard confidence schemes use a confidence threshold to divide branches into high- and low-confidence groups. If the confidence value is above the threshold, the prediction is followed; otherwise, the issuing thread stalls until the branch is resolved. Hard confidence uses ICOUNT to select among the high confidence threads, so the confidence threshold controls how significantly ICOUNT affects fetch. Low thresholds leave the choice almost entirely to ICOUNT, because most threads will be high confidence. High thresholds reduce its influence by providing fewer threads from which to select.

Using hard confidence has two effects. First, it reduces the number of wrong-path-speculative instructions by keeping the processor from speculating on some incorrect predictions (i.e., true negatives). Second, it increases the number of correct predictions the processor ignores (false negatives).

Table III contains true and false negatives for the baseline SMT and an SMT with several hard confidence schemes when executing SPECINT95. Since our MacFarling branch predictor [McFarling 1993] has high accuracy (workload-dependent predictions that range from 88% to 99%), the false negatives outnumber the true negatives by between 3 and 6 times. Therefore, although mispredictions declined by 14% to 88% (data not shown), this benefit was offset by lost successful speculation opportunities, and IPC never rose significantly. In the two cases when IPC did increase by a slim margin (less than 0.5%), JRS and Distance each with a threshold of 1, there were frequent ties among many contexts. Since ICOUNT breaks ties, these two schemes end up being quite similar to ICOUNT.

In contrast to hard confidence, the priority that soft confidence calculates is integrated into the fetch policy. We give priority to contexts that aren't speculating, followed by those fetching past a high confidence branch; ICOUNT breaks any ties. In evaluating soft confidence, we used the same three confidence estimators. Table IV contains the results for SPECINT95. From the table,

Table IV. Soft Confidence Performance for SPECINT95

Confidence Estimator	IPC	Wrong path instructions
No confidence estimation	5.2	9.7%
JRS	5.0	4.5%
Strong	5.0	5.9%
Distance	4.9	2.9%

we see that soft confidence estimators hurt performance, despite the fact that they reduced wrong-path-speculative instructions to between 0.1% and 9% of instructions fetched.

Overall, then, neither hard nor soft confidence estimators improved SMT performance, and actually reduced performance in most cases.

3.2 Why Restricting Speculation Hurts SMT Performance

SMT derives its performance benefits from fetching and executing instructions from multiple threads. The greater the number of active hardware contexts, the greater the global (cross-thread) pool of instructions available to hide intra-thread latencies. All the mechanisms we have investigated that restrict speculation do so by eliminating certain threads from consideration for fetching during some period of time, either by assigning them a low priority or excluding them outright.

The consequence of restricting the pool of fetchable threads is a less diverse thread mix in the instruction queue, where instructions wait to be dispatched to the functional units. When the IQ holds instructions from many threads, the chance of a large number of them being unable to issue instructions is greatly reduced, and SMT can best hide intra-thread latencies. However, when fewer threads are present, it is less able to avoid these delays.[2]

SMT with ICOUNT provides the highest average number of threads in the IQ for all four workloads when compared to any of the alternative fetch policies or confidence estimators. Executing SPECINT95 with soft confidence can serve as a case in point. With soft confidence, the processor tends to fetch repeatedly from threads that have high confidence branches, filling the IQ with instructions from a few threads. Consequently, there are no issuable instructions between 2.8% and 4.2% of the time, which is 3 to 4.5 times more often than with ICOUNT. The result is that the IQ backs up more often (12 to 15% of cycles versus 4% with ICOUNT), causing the processor to stop fetching. This also explains why none of the new policies improved performance—they all reduced the number of threads represented in the IQ. In contrast to all these schemes, ICOUNT works directly toward maintaining a good mix of instructions by favoring underrepresented threads.

We attempted to accentuate this aspect of ICOUNT by modifying it to bound the number of instructions in the IQ from each thread, but instruction diversity and thus performance were unchanged. In fact, even perfect confidence

---

[2]The same effect was observed in Tullsen et al. [1996] for the BRCOUNT and MISSCOUNT policies. These policies use the number of the thead-specific branches and cache misses, respectively, to assign priority. Neither performed as well as ICOUNT.
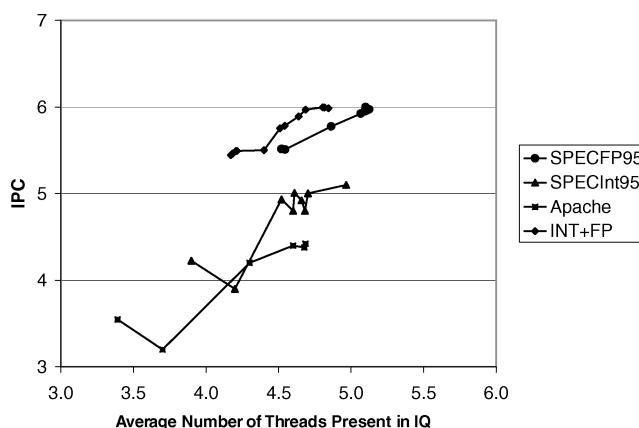
Fig. 5.   The relationship between the average number of threads in the instruction queue and over-all SMT performance. Each point represents a different fetch policy. The relative ordering from left to right of fetch policies differs between workloads. For SPECINT95, no speculation performed worst; the soft confidence schemes were next, followed by the distance estimator (thresh = 3), the strong count schemes, and favoring non-speculative contexts. The ordering for SPECINT+FP is the same. For SPECFP95, soft confidence and favoring nonspeculative contexts performed worst, followed by no speculation and strong count, distance, and JRS hard confidence estimators. Finally, for Apache, soft confidence outperformed no speculation (the worst) and the hard confidence distance estimator but fell short of the hard confidence JRS and strong count estimators. For all four workloads, SMT with ICOUNT is the best performer, although, for SPECINT95 and SPECINT+FP, the hard distance estimator (thresh = 1) obtains essentially identical performance.

estimation (i.e., the processor speculates if the branch prediction is correct and stalls if it is incorrect) provides only a 5% improvement over ICOUNT in the number of contexts represented in the IQ.

Figure 5 empirically demonstrates the effect of thread diversity on performance for all the schemes discussed in this paper, on all workloads (see also Tables VII–X). For all four workloads, there is a clear correlation between performance and the number of threads present; ICOUNT achieves the largest value for both metrics[3] in most cases.

We draw two conclusions from this discussion. First, the key to speculation's benefit is its low cost compared to the benefit of the diverse thread mix it provides in the IQ. If branch prediction was less accurate, speculation would be more costly, and the diversity it adds would not compensate for resources wasted on mispeculation. However, as we will see in Figure 6, branch prediction accuracy generally has to be extremely poor to tip the balance against speculation.

Second, although we investigated only a few of the many conceivable speculation-aware fetch policies, there is little hope that a different speculation-aware fetch policy could improve performance. An effective policy would have to avoid significantly altering the distribution of fetched instructions among the threads and, simultaneously, significantly reduce the number of useless

---

[3]The JRS and Distance estimators with thresholds of 1 acheive higher performance by miniscule margins for some of the workloads. See Section 3.1.3.

instructions fetched. Given the accuracy of modern predictors, devising such a mechanism is unlikely.

3.3 Summary

In this section we examine the performance of SMT processors with speculative instruction execution. Without speculation, an 8-context SMT is unable to provide a sufficient instruction stream to keep the processor fully utilized, and performance suffers. Although the fetch policies we examined reduce the number of wrong-path instructions, they also limit thread diversity in the IQ, leading to lower performance when compared to ICOUNT.

4. LIMITS TO SPECULATIVE PERFORMANCE

In the previous section, we showed that speculation benefits SMT performance for our four workloads running on the hardware we simulated. However, speculation will not improve performance in every conceivable environment. The goal of this section is to explore the boundaries of speculation's benefit—to characterize the transition between beneficial and harmful speculation. We do this by perturbing the software workload and hardware configurations beyond their normal limits to see where the benefits of speculative execution begin to disappear.

4.1 Examining Program Characteristics

Three different workload characteristics determine whether speculation is profitable on an SMT processor:

(1) As branch prediction accuracy decreases, the number of wrong-path instructions will increase, causing performance to drop. Speculation will become less useful and at some point will no longer pay off.
(2) As the basic block size increases, branches become less frequent and the number of threads with no unresolved branches increases. Consequently, more nonspeculative threads will be available to provide instructions, reducing the value of speculation. As a result, branch prediction accuracy will have to be higher for speculation to pay off for larger basic block sizes.
(3) As ILP within a basic block increases, the number of unused resources declines, causing speculation to benefit performance less.

Figure 6 illustrates the trade-offs in all three of these parameters. The horizontal axis is the number of instructions between branches, that is, the basic block size. The different lines represent varying amounts of ILP. The vertical axis is the branch prediction accuracy required for speculation to pay off for a given average basic block size[4]; that is, for any given point, speculation will pay off for branch prediction accuracy values above the point but hurt performance

---

[4]The synthetic workload for a particular average basic block size contained basic blocks of a variety of sizes. This helps to make the measurements independent of Icache block size, but does not remove all the noise due to Icache interactions (for instance, the tail of ILP 1 line goes down).
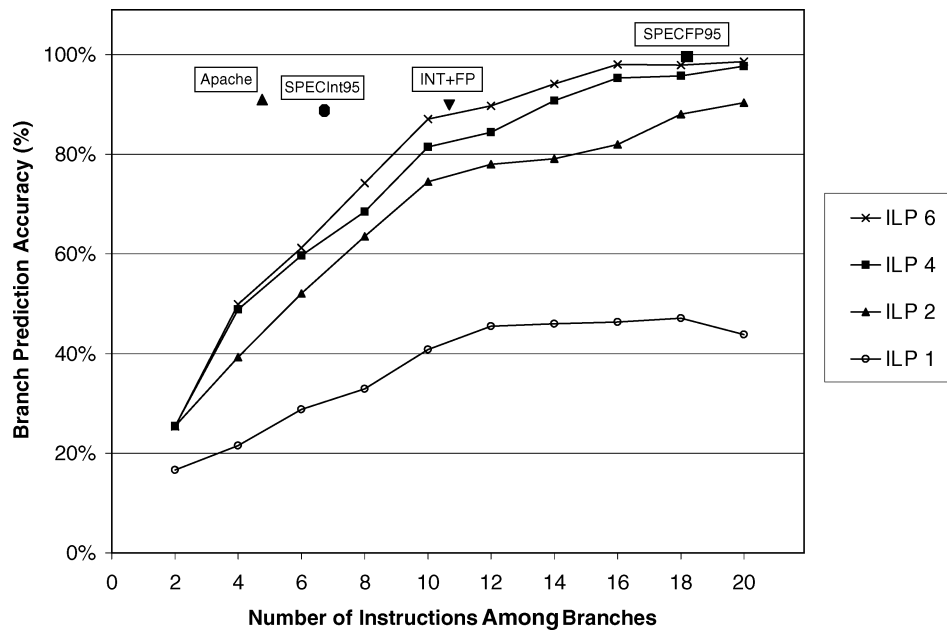
Fig. 6.   Branch prediction accuracies at which speculating makes no difference.

for values below it. The higher this crossover point, the less benefit speculation provides. The data was obtained by simulating a synthetic workload (as described in Section 2.2) on the baseline SMT with ICOUNT (Section 2.1). For instance, a thread with an ILP of 4 and a basic block size of 16 instructions could issue all instructions in 4 cycles, while a thread with an ILP of 1 would need 16 cycles; the former workload requires that branch prediction accuracy be worse than 95% in order for speculation to hurt performance; the latter (ILP 1) requires that it be lower than 46%.

The four labeled points represent the average basic block sizes and branch prediction accuracies for SPECINT95, SPECFP95, INT+FP, and Apache on SMT with ICOUNT. SPECINT95 has a branch prediction accuracy of 88% and 6.6 instructions between branches. According to the graph, such a workload would need branch prediction accuracy to be worse than 65% for speculation to be harmful. Likewise, given the same information for SPECFP95 (18.2 instructions between branches,[5] 99% prediction accuracy), INT+FP (10.5 instructions between branches, 90% prediction accuracy) and Apache (4.9 instructions between branches, 91% prediction accuracy), branch prediction accuracy would have to be worse than 98%, 88% and 55%, respectively. SPECFP95 comes close to hitting the crossover point; this is consistent with the relatively smaller performance gain due to speculation for SPECFP95 that we saw in Section 3.

---

[5]Compiler optimization was set to -O5 on Compaq's F77 compiler, which unrolls loops below a certain size (100 cycles of estimated execution) by a factor of four or more. SPECFP benchmarks have large basic blocks due to both unrolling and to large native loops in some programs.

Similarly, Apache's large distance from its crossover point coincides with the large benefit speculation provides.

The data in Figure 6 show that for modern branch prediction hardware, only workloads with extremely large basic blocks and high ILP benefit from not speculating. While some scientific programs may have these characteristics, most integer programs and operating systems do not. Likewise, it is doubtful that branch prediction hardware (or even static branch prediction strategies) will exhibit poor enough performance to warrant turning off speculation with basic block sizes typical of today's workloads. For example, our simulations of SPECINT95 with a branch predictor one-sixteenth the size of our baseline predictor correctly predicts only 70% of branches, but still experiences a 9.5% speedup over not speculating.

4.2 Examining Hardware Characteristics

We examine three modifications to the SMT hardware that affect how speculation behaves: the number of hardware contexts, the number of functional units, and the size of the level-one caches. While some of these are aggressive, they provide insights into design options and trade-offs surrounding the SMT microarchitecture and illuminate the boundaries of speculation performance. The more conservative configurations are representative of machines that already exist, for example, Marr et al. [2002]; Hinton et al. [2001], or might be built in the near future.

4.2.1 *Varying the Number of Hardware Contexts.* Increasing the number of hardware contexts (while maintaining the same number and mix of functional units and number of issue slots) will increase the number of independent and nonspeculative instructions, and thus will decrease the likelihood that speculation will benefit SMT. Conversely, reducing the number of contexts should increase speculation's value.

One metric that illustrates the effect of increasing the number of hardware contexts is the number of cycles between two consecutive fetches from the same context, or fetch-to-fetch delay. As the fetch-to-fetch delay increases, it becomes more likely that the branch will resolve before the thread fetches again. This causes individual threads to speculate less aggressively, and makes speculation less critical to performance. For a superscalar, the fetch-to-fetch delay is 1.4 cycles. For an 8-context SMT with ICOUNT, the fetch-to-fetch delay is 5.0 cycles—3.6 times longer.

We can use fetch-to-fetch delay to explore the effects of varying the number of contexts in our baseline configuration. With 16 contexts (running two copies of each of the 8 SPECINT95 programs), the fetch-to-fetch delay rises to 10.0 cycles (3 cycles longer than the branch delay), and the difference between IPC with and without speculation falls from 24% for 8 contexts to 0% with 16 (see Figure 7), signaling the point at which speculation should start hurting SMT performance.

At first glance, 16-context non-speculative SMTs might seem unwise, since single-threaded performance still depends heavily on speculation. However, recent chip multi-processor designs, such as Piranha [Barroso et al. 2000], make
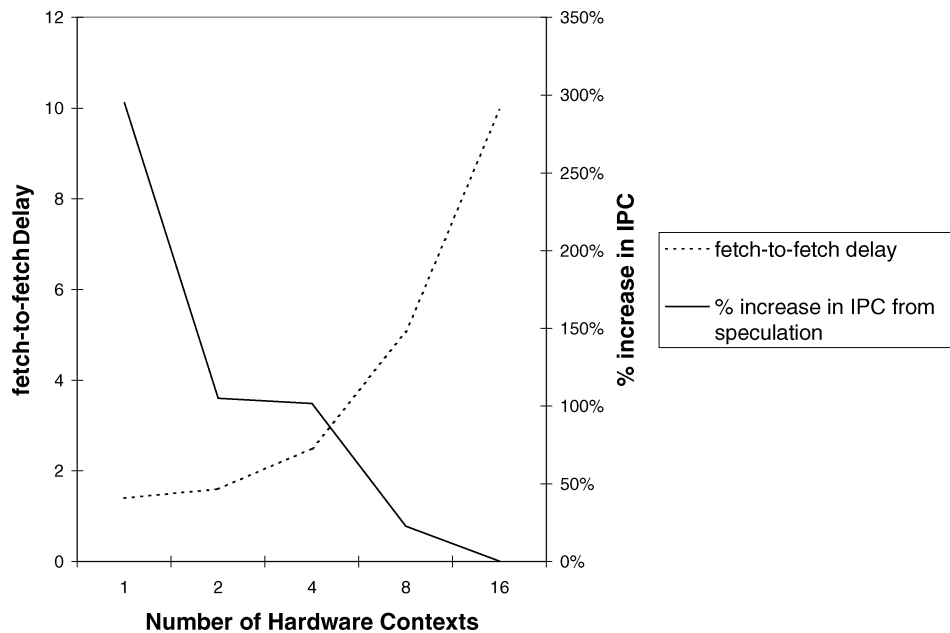
Fig. 7. The relationship between fetch-to-fetch delay and performance improvement due to speculation.

a persuasive argument that single-threaded performance could be sacrificed in favor of a simpler, thoughput-oriented design. In this light, a 16-context SMT might indeed be a reasonable machine to build, despite the complexity of its dynamic issue logic. Not only would it eliminate the speculative hardware, but the large number of threads would make it much easier to hide the large memory latency often associated with server workloads.

Still, forthcoming SMT architectures will most likely have a higher, rather than a lower, ratio of functional units to hardware contexts than even our SMT prototype, which has 6 integer units and 8 contexts. For example, the recently canceled Compaq 21464 [Emer 1999] would have been an 8-wide machine with only four contexts, suggesting that speculation would provide much of its performance. Supporting this conclusion, our baseline configuration with four contexts has a fetch-to-fetch delay of 2.5, and speculation doubles performance.

The data for the 1-, 2-, and 4-context machines also correspond to an 8-context machine running with fewer than 8 threads. Most workloads, with the exception of heavily loaded servers, may not be able to keep all 8 contexts continuously busy. In these cases, fetch-to-fetch delay will decrease as it did for fewer contexts, and speculation will provide similar benefit.

4.2.2 *Functional Units.* We also varied the number of integer functional units between 2 and 10. In each case, one FU can execute synchronization instructions, while the others can perform loads and stores. All the units execute normal ALU instructions. The machines are otherwise identical to the baseline

Table V. Varying the Number of Functional Units

Integer Functional Units	Speculation		No Speculation		
	Benefit from Speculation	Spec. IPC	IPC	Avg. Branch Delay	FU Utilization
2 (1 Load/Store, 1 Synch)	0%	1.9	1.9	21.1	99%
4 (3 Load/Store, 1 Synch)	8%	3.9	3.6	11.7	90%
6 (5 Load/Store, 1 Synch) (baseline)	29%	5.2	4.2	9.4	70%
8 (7 Load/Store, 1 Synch)	22%	5.4	4.4	8.8	55%
10 (9 Load/Store, 1 Synch)	22%	5.4	4.4	8.8	44%

machine. We ran SPECINT95 with each configuration both with and without speculation.

Table V contains the results. For two functional units speculation has no effect, because there is more than enough nonspeculative ILP available and the pipeline is highly congested (the IQ is full between 46% and 65% of cycles and functional unit utilization is 99%). Benefit from speculation first appears with 4 functional units, as the issue width begins to tax the amount of nonspeculative ILP available, but the benefit does not increase uniformly with issue width.

As the number of FUs rises there are two competing effects. First, the processor needs to fetch more instructions to fill the additional functional units, making speculation more important. Second, the instruction queue drains more quickly, causing the average branch delay to decrease (9.4 cycles with 6 FUs, 8.8 with 8 FUs). As a result, threads on the nonspeculating machines spend less time waiting for branches to resolve and can fetch more often, reducing the cost of not speculating. The result is that speculation provides a 29% performance boost with 6 FUs but only 22% with 8 and 10 FUs, although functional unit utilization is lower (65% with 6 FUs, 55% with 8 FUs, and 44% with 10). As the number of FUs climbs, the scarcity of available ILP will dominate, because the average branch delay will approach a minimum value determined by the pipeline (there are 7 stages between fetch and execute). However, for the range of values we explore here, there is an interesting trade-off between the cost of additional functional units and the complexity cost of speculation. For instance, a nonspeculative machine with 6 functional units outperforms a speculative 4 FU machine by 7% and an 8 FU, nonspeculative machine outperforms the 4 FU configuration by 12%.

4.2.3 *Cache Size.*   The memory hierarchy is a significant source of the latency that speculation attempts to hide. Therefore, the size of the instruction and data caches might affect how important speculation is to SMT performance. To quantify this effect we simulated level-1 data and instruction caches ranging from 16KB to 128KB, with and without speculation. Table VI contains the results.

The data show that increasing the size of the level-1 caches decreases the benefit from speculation. There are two reasons for this: First, larger data caches produce less memory latency that needs to be hidden during execution, and therefore speculation is less necessary for good performance. Second, smaller

Table VI. The Benefits of Speculation with Varying Instruction and Data Cache Sizes

Cache size	Speculative IPC	Non-speculative IPC	Benefit From Speculation
16KB	4.0	3.2	33%
32KB	4.4	3.4	29%
64KB	4.9	3.8	28%
128KB	5.2	4.2	24%

instruction caches reduce the number of contexts that are eligible to fetch, that are not waiting on a cache miss, each cycle. (For 128KB caches, an average of 6.6 contexts are eligible, while for a 16KB cache the number drops to 3.9 contexts.) Frequent instruction cache misses have the same negative effect on the IQ as restricting speculation: for the speculative configurations, the number of contexts represented falls from 5.3 with 128KB caches to 4.9 with 16KB caches.

These results support our conclusion that speculation is desirable in the vast majority of cases.

5. RELATED WORK

Several researchers have explored issues in branch prediction, confidence estimation, and speculation, both on superscalars and multithreaded processors. Others have studied related topics, such as software speculation and value prediction.

Wall [1994] examines the relationships between branch prediction and available parallelism on a superscalar and concludes that good branch predictors can significantly enhance the amount of parallelism available to the processor. Hily and Seznec [1996] investigate the effectiveness of various branch predictors under SMT. They determine that both constructive and destructive interference affect branch prediction accuracy on SMT, but they do not address the issue of speculation.

Golla and Lin [1998] investigate a notion similar to fetch-to-fetch delay and its effect on speculation in the context of fine-grained-multithreading architectures. They find that as instruction queues became larger, the probability of a speculative instruction being on the correct path decreases dramatically. They solve the problem by fetching from several threads and thereby increasing the fetch-to-fetch delay. We investigate the notion of fetch-to-fetch delay in the context of SMT and demonstrate that high fetch-to-fetch delays can reduce the need for speculation.

Jacobson et al. [1996], Grunwald et al. [1998], and Manne et al. [1998] suggest using confidence estimators for a wide variety of applications, including reducing power consumption and moderating speculation on SMT to increase performance. Grunwald et al. [1998] provides a detailed analysis of confidence estimator performance but does not address the loss of performance due to false negatives. We demonstrate that false negatives are a significant danger in hard confidence schemes. Both papers restrict their discussion to confidence estimators that produce strictly high- or low-confidence estimates (by setting thresholds), and do not consider soft confidence.

Wallace et al. [1998] uses spare SMT contexts to execute down both possible paths of a branch. They augment ICOUNT to favor the highest confidence path as determined by a JRS estimator and only create new speculative threads for branches on this path. Although they assume that there are one or more unutilized contexts (while our work focuses on more heavily loaded scenarios), their work complements our own. Both show that speculation pays off when few threads are active, either because hardware contexts are available (their work) or threads are not being fetched (ours). Klauser and Grunwald [1999] demonstrate a similar technique, but they do not restrict the creation of speculative threads to the highest confidence path. Instead, they use a confidence estimator to determine when to create a new speculative thread. Because of this difference in their design, ICOUNT performs very poorly, while a confidence-based priority scheme performs much better.

Seng et al. [2000] examine the effects of speculation on power consumption in SMT. They observe that SMT speculates less deeply past branches, resulting in less power being spent on useless instructions. We examine the effect of hardware configuration on SMT's speculative performance and behavior in more detail and demonstrate the connection between fetch policy, the number of hardware contexts, and how aggressively SMT speculates.

Lo et al. [1997b] investigated the effect of SMT's architecture on the design of several compiler optimizations, including software speculation. They found that software speculation on SMT was useful for loop-based codes, but hurt performance on non-loop applications.

6. SUMMARY

This paper examined and analyzed the behavior of speculative instructions on simultaneous multithreaded processors. Using both multiprogrammed and multithreaded workloads, we showed that:

—speculation is required to achieve maximum performance on an SMT;
—fetch policies and branch confidence estimators that favor nonspeculative execution succeed only in reducing performance;
—the benefits of correct-path speculative instructions greatly outweigh any harm caused by hardware resources going to wrong-path speculative instructions;
—multithreading actually enhances speculation, by reducing the percentage of speculative instructions on the wrong path.

We also showed that multiple contexts provide a significant advantage for SMT relative to a superscalar with respect to speculative execution; namely, by interleaving instructions, multithreading reduces the distance that threads need to speculate past branches. Overall, SMT derives its benefit from this fine-grained interleaving of instructions from multiple threads in the IQ. Therefore, policies that reduce the pool of participating threads (e.g., to favor nonspeculating threads) tend to reduce performance.

These results hold for a broad range of hardware configurations and workloads. Only machines with a very high ratio of contexts to issue slots and

functional units or workloads with very large basic block sizes warranted reducing or eliminating speculation. Our results demonstrate that there are interesting microarchitectural trade-offs between speculation, implementation complexity, and single-threaded performance that make the decisions of how and when to speculate on SMT processors more complex than they are on traditional superscalar processors.

APPENDIX

Tables VII–X contain a summary of data for all the fetch policies we investigated.

Table VII. Summary of SPECINT95 Results for all Speculation Schemes

	Fetch IPC		Execute IPC			
		Wrong-path Instructions		Wrong-path Instructions	Commit	Contexts
Fetch Policy	Total	(% of total)	Total	(% of total)	IPC	in IQ
Baseline	6.2	0.7 (10.9)	5.3	0.09 (1.8)	5.2	5.8
Distance; hard (threshold = 1)	6.2	0.7 (10.9)	5.3	0.08 (1.8)	5.2	5.2
Distance; hard (threshold = 3)	5.8	0.3 (4.8)	5.2	0.06 (1.5)	5.1	4.9
Distance; hard (threshold = 7)	5.4	0.2 (3.5)	5.0	0.04 (0.8)	5.0	4.6
Distance; soft	5.5	0.2 (3.6)	5.1	0.05 (1.0)	4.9	4.5
Favor non-speculating contexts	5.9	0.4 (7.4)	5.2	0.6 (1.2)	5.1	4.9
JRS; hard (threshold = 1)	6.1	0.5 (8.9)	5.3	0.1 (1.6)	5.2	5.2
JRS; hard (threshold = 15)	4.9	0.1 (1.1)	4.7	<0.1 (0.2)	4.6	4.6
JRS; soft	5.6	0.3 (5.1)	5.1	0.1 (1.2)	5.0	4.6
Strong; hard (threshold = 1)	5.3	0.6 (11.4)	4.9	0.2 (3.5)	4.8	4.7
Strong; hard (threshold = 2)	4.3	0.2 (4.0)	3.9	0.5 (12.7)	3.8	4.2
Strong; soft	5.7	0.3 (5.8)	5.1	0.1 (1.3)	5.0	4.7
No speculation	4.5	0.0 (0.0)	4.2	0.0 (0.0)	4.2	3.9

Table VIII. Summary of all Apache Results

	Fetch IPC		Execute IPC			
		Wrong-path Instructions		Wrong-path Instructions	Commit	Contexts
Fetch Policy	Total	(% of total)	Total	(% of total)	IPC	in IQ
Baseline	5.2	0.6 (11.0)	4.6	0.1 (2.0%)	4.5	4.9
Distance; hard (threshold = 1)	5.1	0.5 (10.5)	4.5	0.1 (2.0)	4.4	4.7
Distance; hard (threshold = 3)	5.1	0.5 (10.5)	4.5	0.1 (2.0)	4.4	4.7
Distance; hard (threshold = 7)	5.1	0.5 (10.5)	4.5	0.1 (2.0)	4.4	4.7
Distance; soft	5.0	0.5 (9.7)	4.5	0.1 (1.7)	4.4	4.7
Favor non-speculating contexts	5.1	<0.01 (0.1)	5.1	<0.01 (0)	5.1	7.1
JRS; hard (threshold = 1)	5.1	0.5 (10.7)	4.5	0.1 (2.1)	4.4	4.7
JRS; hard (threshold = 15)	5.1	0.5 (10.7)	4.5	0.1 (2.1)	4.4	4.7
JRS; soft	5.1	0.5 (8.9)	4.5	0.1 (1.6)	4.4	4.7
Strong; hard (threshold = 1)	5.0	0.5 (9.9)	4.5	0.1 (1.8)	4.5	4.8
Strong; hard (threshold = 2)	5.0	0.5 (9.9)	3.3	0.1 (3.0)	3.2	3.7
Strong; soft	4.8	0.4 (8.1)	4.4	0.1 (1.8)	4.3	4.4
No speculation	3.5	0.0 (0.0)	3.4	0.0 (0.0)	3.4	3.5

Table IX. Summary of all SPECFP Results

| Fetch Policy | Fetch IPC | | Execute IPC | | Commit IPC | Contexts in IQ |
	Total	Wrong-path Instructions (% of total)	Total	Wrong-path Instructions (% of total)		
Baseline	6.3	0.1 (1.5)	6	0.01 (0.3)	5.9	5.1
Distance; hard (threshold = 1)	6.3	0.06 (1.1)	6.0	0.2 (0.3)	5.9	5.1
Distance; hard (threshold = 3)	6.3	0.06 (1.1)	6.0	0.1 (0.2)	5.9	5.1
Distance; hard (threshold = 7)	6.4	0.06 (1.1)	6.0	0.1 (0.2)	6.0	5.1
Distance; soft	5.8	<0.01 (0.0)	5.5	<0.01 (0.0)	5.5	4.5
Favor non-speculating contexts	5.8	<0.01 (0.0)	5.5	<0.01 (0.0)	5.5	4.5
JRS; hard (threshold = 1)	6.3	0.1 (1.5)	6.0	0.01 (0.3)	6.0	5.1
JRS; hard (threshold = 15)	6.3	0.1 (1.3)	6.0	0.01 (0.3)	6.0	5.1
JRS; soft	5.8	<0.01 (0.0)	5.5	<0.01 (0.0)	5.5	4.5
Strong; hard (threshold = 1)	6.3	0.05 (0.8)	6.0	<0.01 (0.1)	6.0	5.1
Strong; hard (threshold = 2)	5.1	0.3 (0.6)	5.8	<0.01 (0.2)	5.8	4.9
Strong; soft	5.8	<0.01 (0.0)	5.5	<0.01 (0.0)	5.5	4.5
No speculation	5.8	.00 (0.0)	5.5	0.0 (0.0)	5.5	4.5

Table X. Summary of all INT+FP Results

| Fetch Policy | Fetch IPC | | Execute IPC | | Commit IPC | Contexts in IQ |
	Total	Wrong-path Instructions (% of total)	Total	Wrong-path Instructions (% of total)		
Baseline	6.7	0.3 (5.0)	6.0	0.05 (0.8)	6.0	4.8
Distance; hard (threshold = 1)	6.6	0.2 (2.8)	6.0	0.03 (0.5)	6.0	4.8
Distance; hard (threshold = 3)	6.4	0.1 (1.4)	5.9	0.02 (0.3)	6.0	4.6
Distance; hard (threshold = 7)	5.9	0.02 (0.3)	5.5	<0.01 (0.1)	5.9	4.2
Distance; soft	5.9	0.01 (0.1)	5.5	<0.01 (0.0)	5.5	4.2
Favor non-speculating contexts	5.9	0.02 (0.3)	5.5	<0.01 (0.0)	5.5	4.2
JRS; hard (threshold = 1)	6.7	0.3 (3.9)	6.0	0.04 (0.7)	6.0	4.8
JRS; hard (threshold = 15)	6.3	0.08 (1.3)	5.8	0.02 (0.0)	5.8	4.5
JRS; soft	5.9	0.01 (0.2)	5.5	<0.01 (0.0)	5.5	4.2
Strong; hard (threshold = 1)	5.9	0.01 (0.2)	5.6	<0.01 (0.0)	5.5	4.4
Strong; hard (threshold = 2)	6.3	0.1 (1.6)	5.7	0.02 (0.3)	5.8	4.5
Strong; soft	6.3	0.1 (1.6)	5.8	0.02 (0.3)	5.5	4.5
No speculation	5.8	0.0 (0.0)	5.5	0.0 (0.0)	5.5	4.2

REFERENCES

AGARWAL, A., HENNESSY, J., AND HOROWITZ, M. 1988. Cache performance of operating system and multiprogramming workloads. *ACM Trans. Comput. Syst. 6*, 4 (Nov.), 393–431.

BARROSO, L. A., GHARACHORLOO, K., MCNAMARA, R., NOWATZYK, A., QADEER, S., SANO, B., SMITH, S., STETS, R., AND VERGHESE, B. 2000. Piranha: A scalable architecture based on single-chip multi-processing. In *Proceedings of the 27th ACM International Symposium on Computer Architecture*. Vancouver, Canada, 282–293.

COMPAQ. 1998. SimOS-Alpha. http://www.research.digital.com/wrl/projects/SimOS/.

CVETANOVIC, Z. AND KESSLER, R. E. 2000. Performance analysis of the Alpha 21264-based Compaq ES40 system. In *Proceedings of the 27th ACM International Symposium on Computer Architecture*. Vancouver, Canada, 192–202.

EMER, J. S. 1999. Simultaneous multithreading: Multiplying alpha's performance. In *Proceedings of the 1999 International Microprocessor Forum 1999*. MicroDesign Resources, San Jose, California.

GLOY, N., YOUNG, C., CHEN, J. B., AND SMITH, M. D. 1996. An analysis of dynamic branch prediction schemes on system workloads. In *Proceedings of the 23rd ACM International Symposium on Computer Architecture*. Philadelphia, Pennsylvania, 12–21.

GOLLA, P. N. AND LIN, E. C. 1998. A comparison of the effect of branch prediction on multithreaded and scalar architectures. *ACM SIGARCH Comput. Arch. News 26*, 4, 3–11.

GRUNWALD, D., KLAUSER, A., MANNE, S., AND PLESZKUN, A. 1998. Confidence estimation for speculation control. In *Proceedings of the 25th ACM International Symposium on Computer Architecture*. Barcelona, Spain, 122–131.

GWENNAP, L. 1995. New algorithm improves branch prediction. *Microprocessor Reports*, 17–21.

HILY, S. AND SEZNEC, A. 1996. Branch prediction and simultaneous multithreading. In *Proceedings of Parallel Architectures and Compilation Techniques (PACT 96)*. 196–173.

HINTON, G., SAGER, D., UPTON, M., BOGGS, D., CARMEAN, D., KYKER, A., AND ROUSSEL, P. 2001. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal 5*, 1 (Feb.).

HU, Y., NANDA, A., AND YANG, Q. 1999. Measurement, analysis and performance improvement of the Apache web server. In *Proceedings of the 18th IEEE International Performance, Computing and Communications Conference (IPCCC'99)*. Phoenix/Scottsdale, AZ, 261–267.

JACOBSON, E., ROTENBERG, E., AND SMITH, J. E. 1996. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th IEEE/ACM International Symposium on Microarchitecture*. Paris, France, 142–152.

KLAUSER, A. AND GRUNWALD, D. 1999. Instruction fetch mechanisms for multipath execution processors. In *Proceedings of the 32nd IEEE/ACM International Symposium on Microarchitecture*. 38–47.

LO, J. L., EGGERS, S. J., EMER, J. S., LEVY, H. M., STAMM, R. L., AND TULLSEN, D. M. 1997a. Converting thread-level parallelism into instruction-level parallelism via simultaneous multithreading. *ACM Trans. Comput. Syst. 15*, 3 (Aug.), 322–354.

LO, J. L., EGGERS, S. J., LEVY, H. M., PAREKH, S., AND TULLSEN, D. M. 1997b. Tuning compiler optimizations for simultaneous multithreading. In *Proceedings of the 30th IEEE/ACM International Symposium on Microarchitecture*. Research Triangle Park, North Carolina, 114–124.

MANNE, S., KLAUSER, A., AND GRUNWALD, D. 1998. Pipeline gating: Speculation control for energy reduction. In *Proceedings of the 25th ACM International Symposium on Computer Architecture*. Barcelona Spain, 132–141.

MARR, D. T., BINNS, F., HILL, D. L., HINTON, G., KOUFATY, D. A., MILLER, J. A., AND UPTON, M. 2002. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal 6*, 1 (Feb.).

MCFARLING, S. 1993. Combining branch predictors. Tech. Rep. Technical Note TN-36, Digital Equipment Corporation, Western Research Lab. June.

REDSTONE, J. A., EGGERS, S. J., AND LEVY, H. M. 2000. An analysis of operating system behavior on a simultaneous multithreaded architecture. In *Proceedings of the Ninth ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Cambridge, Massachusetts, 245–256.

REILLY, J. 1995. SPEC describes SPEC95 products and benchmarks. *SPEC Newsletter*. http://www.specbench.org/.

ROSENBLUM, M., HERROD, S. A., WITCHEL, E., AND GUPTA, A. 1995. Complete computer simulation: The SimOS approach *4*, 3 (Winter), 34–43.

SENG, J. S., TULLSEN, D. M., AND CAI, G. Z. N. 2000. Power-sensitive multithreaded architecture. In *Proceedings of the 2000 International Conference on Computer Design*. 199–206.

SNAVELY, A. AND TULLSEN, D. M. 2000. Symbiotic job scheduling for a simultaneous multithreaded processor. In *Proceedings of the Ninth ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Cambridge, Massachusetts, 234–244.

SYSTEM PERFORMANCE EVALUATION COOPERATIVE, S. 1996. An explanation of the SPECWeb96 benchmark. http://www.specbench.org/osg/web96/.

SYSTEM PERFORMANCE EVALUATION COOPERATIVE, S. 2000. Run and reporting rules for SPEC CPU2000. http://www.specbench.org/osg/cpu2000/.

TULLSEN, D. M. 1996. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conference*. 819–828.

TULLSEN, D. M., EGGERS, S. J., EMER, J. S., LEVY, H. M., LO, J. L., AND STAMM, R. L. 1996. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd ACM International Symposium on Computer Architecture*. 191–202.

TULLSEN, D. M., EGGERS, S. J., AND LEVY, H. M. 1995. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd ACM International Symposium on Computer Architecture*. Santa Margherita Ligure, Italy, 392–403.

WALL, D. W. 1994. Speculative execution and instruction-level parallelism. Tech. Rep. Technical note TN-42, Digital Equipment Corporation, Western Research Lab. Mar.

WALLACE, S., CALDER, B., AND TULLSEN, D. M. 1998. Threaded multiple path execution. In *Proceedings of the 25th ACM International Symposium on Computer Architecture*. Barcelona Spain, 238–249.