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

Abstract

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-wide 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, in fact, critical to performance on a multithreaded processor, and (2) SMT actually *enhances* the effectiveness of speculative execution, compared to a superscalar processor.

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) [8], 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 are often underutilized on superscalars because of low single-thread instruction-level parallelism (ILP) [28, 3], 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 [28, 27] 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 [16, 15, 20].

With its high hardware utilization, speculation on an SMT may *harm* rather than improve performance, particularly with all hardware contexts occupied. Speculative (and potentially wasteful) instructions from one thread compete with useful, non-speculative instructions from other threads for highly utilized hardware resources, and in some cases displace them, lowering performance. Therefore, it is important to understand the behavior of speculation on an SMT processor and the extent to which it helps or hinders performance.

This paper investigates the interactions between instruction speculation and multithreading

and quantifies the impact of speculation on the performance of SMT processors. 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 SMT performance 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 speculation affects, 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 increasing the number of hardware threads and using synthetic workloads to change the branch frequency and ILP within threads.

Our results show that despite its potential downside, speculation increases SMT performance by 9% to 32% over a non-speculating SMT. Furthermore, modulating speculation with either a speculation-aware fetch policy or confidence estimation never improved, and usually lowered, performance. Two separate factors explain this behavior. First, SMT processors actually *rely* on speculation to provide a rich, cross-thread instruction mix to fully utilize the functional units. While sophisticated speculation policies can reduce speculation's negative effects, they simultaneously restrict the diversity of threads occupying the pre-execution stages of the pipeline. Consequently, SMT only benefits from less aggressive speculation when executing unlikely workloads or on extremely aggressive hardware designs. Second, and somewhat surprisingly, SMT *enhances* the effectiveness of speculation by decreasing the percentage of speculative instructions on the wrong path. SMT reduced the percentage of wrong-path speculative instructions by 29% when compared to a superscalar with comparable execution resources.

The remainder of this paper is organized as follows. The next section details the methodology for our experiments. Section 3 presents the basic speculation results and explains why and how speculation benefits SMT performance; it also presents alternative fetch and prediction schemes and shows why they fall short. Section 4 explores 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 [26] and has been ported to the SimOS framework [22, 2, 20]. 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 1. For most experiments we used the ICOUNT fetch policy [27]. 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 up to 8 to issue, 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, i.e., 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 with speculative instructions; ties are broken with ICOUNT. The third uses branch confidence estimators to favor threads with high-confidence branches.

Our baseline experiments used the McFarling branch prediction algorithm [19] used on modern processors from Compaq; for some studies we augmented this with confidence estimators.

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 [27])
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 [19] (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

Table 1: SMT parameters.

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 [21], and a combination of four applications from each suite, INT+FP. In addition we used the Apache web server, an open source web server run by the majority of web sites [10]. We drive Apache with SPECWEB96 [25], a standard web server performance benchmark. 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 [1, 5], 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 less frequent. Finally, following the example of Snavely and Tullsen [34], 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 across experiments, 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; longer simulations had no significant effect on 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 can resolve this problem by following the example set by the SPECrate metric [24] 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 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. We then 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 the branch) and compared the throughput in instructions per cycle of our four workloads on the speculative and non-speculative SMT CPUs. The results of these measurements are shown in Table 2. Speculation benefits SMT performance on all four workloads – the speculative SMT achieves performance gains of between 9% and 32% over the non-speculative processor.

	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%

Table 2: Effect of speculation on SMT. We simulated each of the four workloads on machines with and without speculation. Apache, with its small basic blocks and poor branch prediction, derives the most performance from speculation, while the more predictable floating benchmarks benefit least.

Speculation can have 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 simulation. Changing the speculation policy never alters the percentage of

cache hits by more that 0.4%.

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

- **non-speculative** instructions are those fetched non-speculatively, i.e., 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, and 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-/wrong-path-/non-speculative per-stage IPCs. The overall machine IPC is the sum of the correct-path-speculative and non-speculative commit IPCs. Figure 1 depicts the per-stage instruction categories for SPECINT95.[1]

Figure 1a shows why speculation is crucial to high instruction throughput and explains why misspeculation does not waste hardware resources. Speculative instructions on an SMT comprise the majority of instructions fetched, executed and committed: for SPECINT95, 57% of fetch IPC, 53% of instructions issued to the functional units, and 52% of commit IPC are speculative. Given the magnitude of these numbers and the accuracy of today's branch prediction hardware, it would be extremely surprising if ceasing to speculate or significantly reducing the number of speculative instructions improved performance.

Speculation is particulary effective on SMT for two reasons. First, since SMT fetches from

[1]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, e.g., the same fraction of wrong-path speculative instructions reach the execute stage for all 4 workloads. Furthermore, 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 only present data for SPECINT95 in many of the figures.

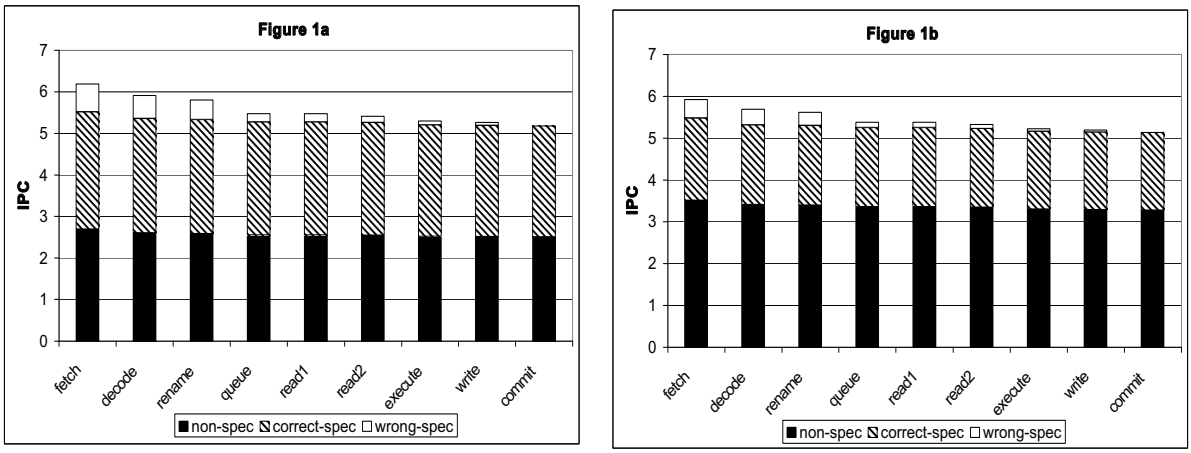

Figure 1. Per-pipeline-stage IPC for SPECINT95, divided between correct-path-, wrong-path-, and non-speculative instructions. On the left, (a) SMT with ICOUNT; on the right, (b) SMT with a fetch policy that favors non-speculative instructions.

each thread only once every 5.4 cycles on average (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 in this experiment was 88%[1] and only 11% of fetched instructions were flushed from the pipeline. 73% of the 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

[1]The prediction rate is lower than the value found in [8] because we include the operating system code.

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 size 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. [27] reports 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 Section 3.2.

Without speculation, only non-speculative instructions use processor resources and SMT devotes no processor resources to wrong-path instructions. However, in avoiding wrong-path instructions, SMT leaves many of its hardware resources idle. Fetch stall cycles, i.e., cycles when no thread was fetched, rose almost five-fold; consequently, per stage IPCs dropped between 19% and 29%. Functional utilization dropped by 20% and commit IPC, the bottom-line metric for SMT performance, was 4.2, a 19% loss compared to an SMT that speculates. Contrary to the usual hypthesis, not speculating wastes more resources than speculating!

3.2 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 non-speculative instructions. To investigate these possibilities, we compared SMT with ICOUNT to an SMT with two alternative fetch policies: one that favors non-speculating threads and a family of fetch policies that incorporate branch prediction confidence.

3.2.1 Favoring non-speculative contexts

A fetch policy that favors non-speculative contexts (see Figure 1b) increased the proportion of non-speculative 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 3.3.

3.2.2 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 [11, 7]. 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 [11, 18]. In our study we examined three different confidence estimators discussed in [7, 11]:

- 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 [7]).
- 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 context has made in a row (globally, not just for this branch).

There are (at least) two different ways to use such confidence information. In the first, *hard confidence*, the processor stalls the thread on a low confidence branch, fetching from other threads until the branch is resolved. In the second, *soft confidence*, the processor assigns priority according to confidence of the 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. 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 3 contains true and false negatives for the baseline SMT and an SMT with several hard

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

Table 3: Hard confidence performance for SPECINT95. Branch prediction accuracy was 88%.

confidence schemes when executing SPECINT95 (SPECFP95, INT+FP, and Apache had similar results). Since our MacFarling branch predictor [19] 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 between 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 4 contains the results for SPECINT95. From the table, we see that soft confidence estimators hurt performance, despite the fact that they reduced wrong-

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%

Table 4: Soft confidence performance for SPECINT95.

path-speculative instructions from 9% to between 2.9% and 5.9% of instructions fetched.

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

3.3 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. [1]

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

[1]The same effect was observed in [27] for the BRCOUNT and MISSCOUNT policies. These policies use the number of thread-specific outstanding branches and cache misses, respectively, to assign fetch priority. Neither performed as well as ICOUNT.

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 front end of 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, ICOUNT works directly toward a good mix of instructions by favoring underrepresented threads.

Figure 2 correlates the average number of threads represented in the IQ with the IPC of all schemes discussed in this paper, on all workloads. 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[1] in most cases.

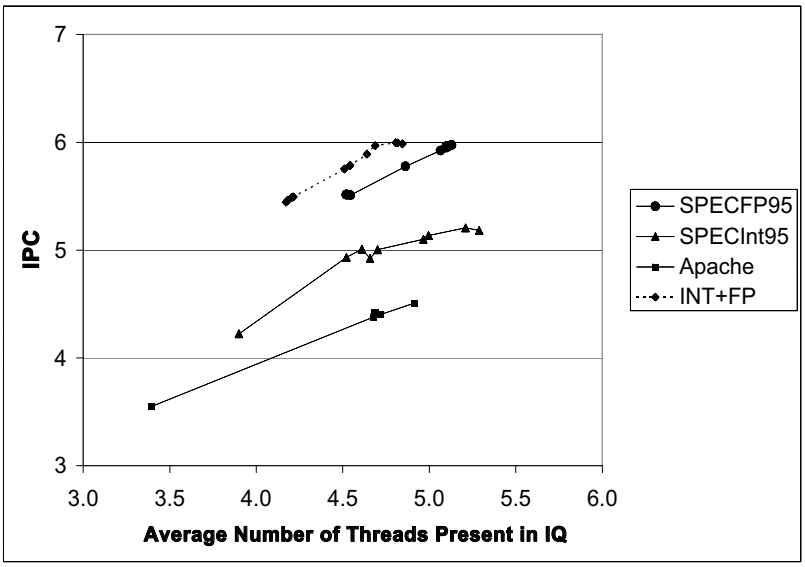

Figure 2. The relationship between the average number of threads in the instruction queue and overall SMT performance. Each point represents a different fetch policy. The relative ordering from left to right of fetch policies differs between workloads. For SPECINT95, no speculation performed worst; the 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 non-speculative 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.

[1]The JRS and Distance estimators with thresholds of 1 achieve higher performance by minuscule margins for some of the workloads. See Section 3.2.2.

3.4 Summary

In this section we examined 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 greatly 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, i.e., 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 non-speculative 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 3 illustrates the trade-offs in all three of these parameters. The horizontal axis is the number of instructions between branches, i.e., the basic-block size. The different lines represent varying amounts of ILP. The vertical axis is the branch prediction accuracy required for specula-

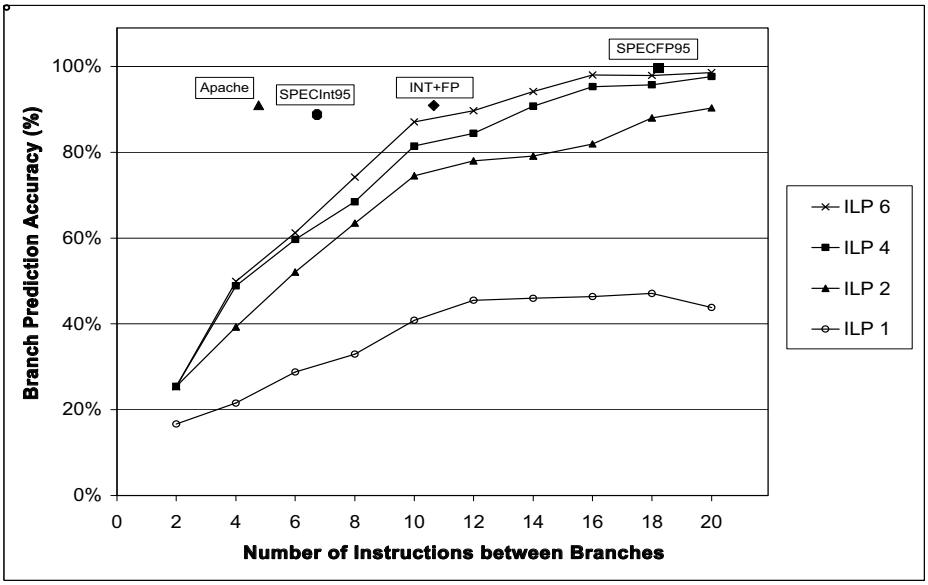

Figure 3. Branch prediction accuracies at which speculating makes no difference.

tion to pay off for a given average basic-block size[1]; that is, for any given point, speculation will pay off for branch prediction accuracy values above the point but hurt performance for values below it. The higher the cross-over 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[2] instructions

[1]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 the ILP1 line goes down).

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

The data in Figure 3 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 Increasing 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 non-speculative instructions, and thus will decrease the likelihood that speculation will benefit SMT. 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.

[2]Compiler optimization was set to -O5 on Compaq's F77 compiler, which unrolls, by a factor of 4 or more, loops below a certain size (100 cycles of estimated execution). SPECFP benchmarks have large basic blocks due both to unrolling and to large native loops in some programs.

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 adding contexts to 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 4), signaling the point at which speculation should start hurting SMT performance.

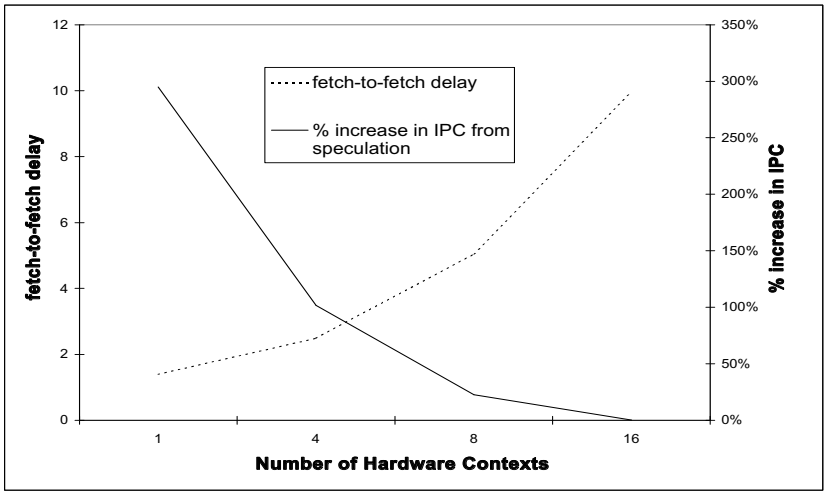

Figure 4. The Relationship Between Fetch-to-Fetch and Performance Improvement Due to Speculation.

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 the Piranha [32] from Compaq, make 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. Not only would it not require the complexity of 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 [4] would have been an 8-wide machine with only four contexts, suggesting that speculation will provide much of its performance.

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 [29] 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 [9] 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 [6] 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.

Jacobsen et al. [11], Grunwald et al. [7], and Manne et. al. [18] suggest using confidence estimators for a wide variety of applications, including reducing power consumption and moderating speculation on SMT to increase performance. Manne et al. [7] provides a detailed analysis of confidence estimator performance but does not address the loss of performance due to false nega-

tives. 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. [30] 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 [12] 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. [33] 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 aggressiveness of SMT in more detail and demonstrate the connection between fetch policy, the number of hardware contexts, and how aggressively SMT speculates.

Lo et. al. [17] 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 non-speculative execution succeed only in reducing performance;
- the benefits of correct-path speculative instructions greatly outweigh any harm caused by wrong-path speculative instructions; and
- 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 non-speculating threads) tend to reduce performance.

These results hold for all "reasonable" hardware configurations and workloads we tested; only extreme conditions (i.e., a very high ratio of contexts to issue slots and functional units or very large basic block sizes) warranted reducing or eliminating speculation. However, there remain 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.

References

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

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

[3] Z. Cvetanovic and R. Kessler. Performance analysis of the Alpha 21264-based Compaq ES40 System. In *27th Annual International Symposium on Computer Architecture*, June 2000.

[4] J. Emer. Simultaneous multithreading: Multiplying Alpha's performance, October 1999.

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

[6] P. Golla and E. Lin. A comparison of the effect of branch prediction on multithreaded and scalar architectures. *ACM SIGARCH Computer Architecture News*, 26(4):3–11, 1998.

[7] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun. Confidence estimation for speculation control. In *25nd Annual International Symposium on Computer Architecture*, June 1998.

[8] L. Gwennap. New algorithm improves branch prediction. *Microprocessor Reports*, March 27 1995.

[9] S. Hily and A. Seznec. Branch prediction and simultaneous multithreading. In *International Conference on Parallel Architecture and Compilation Techniques*, 1996.

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

[11] E. Jacobson, E. Rotenberg, and J. Smith. Assigning confidence to conditional branch predictions. In *29th Annual International Symposium on Microarchitecture*, December 1996.

[12] A. Klauser and D. Grunwald. Instruction fetch mechanisms for multipath execution processors. In *32nd Annual International Symposium on Microarchitecture*, November 1999.

[13] M. Lipasti and J. Shen. Exceeding the dataflow limit via value prediction. In *29th Annual International Symposium on Microarchitecture*, December 1996.

[14] M. Lipasti, C. B. Wilkerson, and J. Shen. Value locality and load value prediction. In *7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[15] 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.

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

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

[18] S. Manne, D. Grunwald, and A. Klauser. Pipeline gating: Speculation control for energy reduction. In *25th Annual International Symposium on Microarchitecture*, June 1998.

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

[20] J. Redstone, S. Eggers, and H. Levy. Analysis of operating system behavior on a simultaneous multithreaded architecture. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

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

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

[23] J. Smith. A study of branch prediction strategies. In *8th Annual International Symposium on Computer Architecture*, May 1981.

[24] SPEC CPU2000. Run and reporting rules for SPEC CPU2000, 2000. http://www.specbench.org/.

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

[26] D. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conference*, December 1996.

[27] 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.

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

[29] D. Wall. Speculative execution and instruction-level parallelism. Technical Report 42, Digital Western Research Laboratory, March 1994.

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

[31] T.-Y. Yeh and Y. Patt. Alternative implementations of two-level adaptive branch prediction. In *19th Annual International Symposium on Computer Architecture*, May 1992.

[32] L. Barroso, et. al. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *27th Annual International Symposium on Computer Architecture*, June 2000.

[33] J. Seng, D. Tullsen and G. Cai. Power-Sensitive Multithreaded Architecture. In *Proceedings of the 2000 International Conference on Computer Design*, 2000.

[34] A. Snavely and D. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor. In *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems,* Nov. 2000