# Aggressive Scheduling and Speculation in Multithreaded Architectures: Is it Worth its Salt?

Jason Loew, Dmitry Ponomarev
*Binghamton University*
*{jloew, dima} @cs.binghamton.edu*

## Abstract

*This paper investigates and quantifies the impacts of several aggressive performance-boosting techniques designed for superscalar processors on the performance of SMT architectures. First, we examine the synergy of multithreading and speculative execution. Second, we quantify the performance impact of not supporting the load-hit speculation. Finally, we consider the impact of pipelining instruction scheduling logic over two cycles. The general conclusion of our studies is that while speculative execution is still important to achieve high SMT performance, scheduler-related mechanisms can be relaxed because the pipeline bubbles created in the execution schedule of one thread are often filled by the instructions from other threads.*

## 1. Introduction

Microarchitectural approaches for extracting Instruction-Level Parallelism (ILP) out of single-threaded applications typically rely on the use of large, complex, highly-speculative out-of-order processing cores. In order to realize high performance, these microprocessors employ a myriad of speculative techniques, ranging from branch prediction [16,13,11] to load-latency prediction [12] to memory dependence prediction [4]. In addition, in single-threaded processors it is imperative to implement aggressive dynamic scheduling mechanisms that allow for the execution of two dependent instructions (both with single-cycle execution latency) in the back-to-back cycles to avoid drastic degradations in the instruction throughput.

Branch prediction is used to predict the outcomes of conditional branches and speculatively execute the instructions along the predicted path without waiting for the actual branch outcome to be resolved. While the datapath complexities are introduced in maintaining the branch prediction tables and, especially, in providing mechanisms for branch misprediction recovery, this form of speculation is essential for single-threaded environments where stalling for branch resolution results in many wasted cycles. Load-hit speculation is used to predict if load-dependent instructions can be speculatively scheduled without waiting for the outcome of the corresponding cache access (hit or miss) to be calculated – this allows the reduction of the load-to-use delay and helps build more efficient and faster execution schedules [22]. Waiting for a load instruction to be serviced from the memory hierarchy introduces a pipeline bubble between the load and load-dependent instructions that adversely impacts performance. Although most load instructions hit into the L1 cache [22], an incorrect prediction can result in replaying of the load and its dependent instructions, wasting critical resources that could be allocated elsewhere. Furthermore, the technique puts additional pressure on the issue queue, as the duration of the instruction residency in the queue increases because the entries allocated to the load-dependent instructions are not released until the load-hit prediction is verified. Finally, high throughput of single-threaded workloads is sustained by guaranteeing that dependent instructions can be executed in the back-to-back cycles. One way to achieve this goal is to implement the basic scheduling activities (such as wakeup and selection operations) in the same clock cycle. However, such a stringent requirement can significantly slow down the processor's clock as the scheduling logic often defines the critical path. An alternative is to implement wakeup and selection activities in separate cycles, and rely on highly-complex speculative techniques to enforce back-to-back execution. [17,2]

While all of the above techniques are important in the domain of superscalar processors, their efficacy on a Simultaneously Multithreaded (SMT) processor is less obvious. SMT architectures provide an area-efficient enhancement to superscalars by multiplexing several independent instruction streams and sharing most of the datapath resources among them. In the presence of explicit Thread-level Parallelism (TLP) through multiple simultaneous threads, the impact of

1550-6533/08 $25.00 © 2008 IEEE
DOI 10.1109/SBAC-PAD.2008.15

aggressive speculation and scheduling techniques needs to be reassessed. For example, in an SMT machine with a sufficient number of threads it may be possible to avoid speculative execution altogether and instead rely on the abundant supply of non-speculative instructions to sustain the instruction throughput. Furthermore, branch mispredictions can adversely impact unrelated threads by causing wrong-path instructions to compete for processor resources with correct-path instructions from other threads. Similarly, load-hit speculation can result in an unnecessary replay of instructions, due to mispredictions, instead of allowing other load-independent instructions to execute. Without load-hit speculation, it could be still possible to generate a high-performance dynamic execution schedule from multiple streams by filling the load-to-use delay slot of some threads with independent instructions from other threads. Likewise, the pipelined implementation of the dynamic scheduling logic may be possible by filling in the bubbles in the execution schedule of one thread by the instructions from other threads. In this paper, we attempt to undertake a comprehensive study of these mechanisms on an SMT processor in an effort to understand the resulting performance nuances and make conclusions about the utility of these techniques on SMT. As the biggest concern in implementing an SMT machine with larger number of threads is in the potential increase in the datapath complexity and power dissipation (as key shared resources have to be enlarged to support many thread contexts), it is important to quantitatively demonstrate that TLP enables various simplifications within the datapath to be realized, primarily in the dynamic scheduling logic. The main contributions and the key results of this work are:

- Using detailed, cycle-accurate microarchitectural simulator of an SMT processor and multiprogrammed workloads of SPEC 2000 benchmarks, we perform a comprehensive evaluation of aggressive scheduling and speculation in multithreaded processors.
- We generally confirm the conclusions made in earlier work [18] that branch prediction is still important for SMT performance, even in the presence of a fairly large number of threads.
- We establish that the performance impact of disabling speculative scheduling (i.e. not using the load hit/miss prediction and instead waiting for the outcomes of the cache accesses to be resolved before scheduling dependent instructions) is significantly smaller on SMT than on a superscalar and it strongly depends on both the number of threads and the depth of the issue-to-execute portion of the pipeline. Consequently, the additional complexity of implementing speculative scheduling on SMT is often not justified.
- Finally, we evaluate the impact of pipelining the scheduling logic over two cycles and thus hindering the ability to execute dependent instructions back-to-back. We evaluate various instruction selection schemes and demonstrate that *distributed selection scheme* (where instructions scheduled together in one cycle are chosen from as many threads as possible) provides performance within 1.6% of the ideal atomic scheduler for 4-threaded workloads. It is thus possible to pipeline the scheduling logic on SMT with almost no impact on processor's performance.

The remainder of this paper is organized as follows: Section 2 discusses related work. Section 3 describes the simulation environment, workloads and metrics used to evaluate all of our techniques. Section 4 quantifies the impact of disabling branch prediction on SMT machines and explores alternative schemes. Section 5 explores the impact of removing speculative scheduling on SMT. Section 6 examines the penalty of removing back-to-back execution and implementing various pipelined selection schemes on SMT and Section 7 summarizes the results.

## 2. Related Work

The only prior research that directly investigated the performance impact of speculative execution on multithreading architectures is the work of [18]. The authors of [18] demonstrated that speculative execution is still important for SMT performance. Our studies generally confirm those conclusions. We also investigate alternative schemes of using branch prediction on SMT. Furthermore, this submission extends the work of [18] by also considering the performance issues associated with speculative scheduling and pipelined implementation of the scheduling logic; none of these were previously evaluated and quantified in the context of SMT machines.

The issues related to branch prediction, latency prediction/speculative scheduling and the back-to-back execution of dependent instructions were extensively investigated in the literature in the context of superscalar processors. The body of work on branch prediction is too large to extensively site here and we refer the readers to [7] for more detailed discussions. The aspects of speculative instruction scheduling and associated scheduling replay schemes were thoroughly described in the work of [9]. Several solutions for relaxing the scheduling loop in superscalar processors to provide the pipelined implementation of the scheduling logic without degrading throughput were proposed. One solution involves speculative wakeup [18] based on the scheduling information about an

instruction's grandparents. Another approach is select-free scheduling [2] that assumes all woken up instructions can be immediately selected for execution and avoids complicated selection logic. Finally, the Macro-OP scheduling mechanism of [8] groups two (or more) dependent instructions together and considers them as one coarser-grain entity for the scheduling purposes. While relaxing the scheduling loop, all of these techniques introduce substantial additional complexities elsewhere within the datapath. In this paper, we demonstrate that such complex solutions are often not needed on SMT, as the presence of explicit TLP allows us to sustain the instruction throughput even with much less aggressive forms of scheduling and speculation.

## 3. Simulation Methodology

To evaluate all of the ideas proposed in this paper we used M-Sim [15] – a cycle-accurate simulator of a detailed SMT pipeline. M-Sim extends Simplescalar by not only supporting the SMT execution model, but also by modeling in a very detailed fashion all key datapath structures (such as the issue queue, the reorder buffer, the register file, the rename table and the load/store queue), explicitly simulating register renaming and accurately modeling the schedule-to-execute pipeline and all activities involved in speculative instruction scheduling. All of these modifications are important for our study. The default configuration used is described in Table 1.

For our experiments, we used a full set of SPEC 2000 [6] benchmarks. The pre-compiled Alpha AXP binaries, as obtainable from the SimpleScalar website, were used. For each test, we first fast forwarded the execution by 100 million instructions and then simulated until the next 100 million instructions committed. With multithreaded workloads, simulations were stopped when 100 million instructions from at least one thread had been committed. The multithreaded workloads were created by first simulating the execution of each benchmark in isolation in a single-threaded execution mode and categorizing them into low-ILP, medium-ILP or high-ILP groups, depending on the performance level. The multithreaded workloads have then been created by mixing the benchmarks with the various ILP levels together. Representative benchmarks from each group were chosen randomly, but we made an effort to include every benchmark available in the SPEC 2000 suite into our evaluations. The resulting multithreaded workloads that were used are as shown in Table 2.

**Table 1: Default SMT Configuration**

| Parameter | Configuration |
|---|---|
| Machine Width | 8-wide fetch, 8-wide issue, 8-wide commit |
| Window Size | 128 entry per-thread ROB, 64 entry shared IQ, 48 entry per-thread LSQ |
| Function Units and Latency (total/issue) | 8 Int Add (1/1), 4 Int Mult/ (3/1) Div (20/19), 4 Load/Store (2/1), 8 FP Add (2/2), 4 FP Mult (4/1)/ Div (12/12)/ Sqrt (24/24) |
| Physical Registers | 512 integer + 512 floating point registers |
| L1 I-Cache | 32KB, 2-way set-associative, 32 byte line, 1 cycle hit time |
| L1 D-Cache | 64KB, 4-way set-associative, 32 byte line, 2 cycle hit time |
| L2 Cache | Unified: 512KB, 8-way set-associative, 12 cycle hit time |
| BTB | 2048 sets, 2-way set associative |
| Branch Predictor | gshare: 4k entries, 10-bit global history per thread |
| Memory | 64 bit wide, 300 cycle first chunk access, 2 cycle interchunk access |
| Load-Hit Predictor | 2-bit bimodal: 1k entries, 8-bit global history per thread. |
| TLB | 128 entry (I), 64 entry (D), fully associative, 30 cycle miss latency |
| Fetch-Rename Delay | 4 cycles |

**Table 2: Multithreaded Workloads**

| 2-Threaded Workloads | | 3-Threaded Workloads | | 4-Threaded Workloads | |
|---|---|---|---|---|---|
| 2-1 | perlbmk, eon | 3-1 | ammp, eon, vpr | 4-1 | perlbmk, mgrid, ammp, parser |
| 2-2 | fma3d, sixtrack | 3-2 | equake, fma3d, gzip | 4-2 | sixtrack, art, apsi, applu |
| 2-3 | bzip2, mcf | 3-3 | lucas, mesa, mcf | 4-3 | gcc, mgrid, vortex, sixtrack |
| 2-4 | gcc, sixtrack | 3-4 | parser, gzip, bzip2 | 4-4 | wupwise, galgel, bzip2, mcf |
| 2-5 | eon, wupwise | 3-5 | mgrid, fma3d, art | 4-5 | sixtrack, fma3d, lucas, mesa |
| 2-6 | gzip, twolf | 3-6 | apsi, vortex, twolf | 4-6 | vpr, applu, galgel, wupwise |
| | | 3-7 | ammp, gcc, mcf | 4-7 | ammp, vpr, mcf, twolf |
| | | 3-8 | vpr, bzip2, swim | 4-8 | parser, perlbmk, equake, lucas |
| | | 3-9 | ammp, parser, fma3d | 4-9 | eon, apsi, facerec, swim |
| | | 3-10 | apsi, swim, lucas | | |

The simulation results are presented using two different metrics. The first metric is throughput IPC, which is the cumulative IPC of all simultaneously executing threads. However, if a technique that is being evaluated favors high-IPC threads then the overall throughput can increase at the expense of hindering the progress of low-IPC threads. This issue is addressed by using a "fairness" metric [10] (or harmonic mean of weighted IPCs).

## 4. Impact of Speculative Execution on SMT

We begin our evaluations in this section by examining the impact of completely disabling speculative execution on SMT and instead forcing the threads that encounter a branch to wait at the front end (and avoid further fetches) until the branch in question is resolved. Just as with the rest of the schemes described in this section, the performance impact will

be directly compared against that observed for a single-threaded superscalar machine. We refer to this mechanism as DISPEX (DIsabled SPeculative EXecution). DISPEX can simply be implemented by disabling branch prediction and removing the associated hardware including the branch misprediction handling logic throughout the pipeline. One additional bit per thread is needed to indicate the status of that thread (whether it is blocked or not). The performance impact of using DISPEX on the evaluated SMT machine is shown in Table 3. Results are presented as performance losses (in terms of both throughput and fairness for SMT machines) with respect to the corresponding machine with speculative execution. As seen from the results, even with large number of thread contexts, the performance losses are still very significant, the fact corroborated by previous work [18]. The main reason for such drastic degradation in the absence of control flow speculation is the lack of a mix of instructions from multiple threads to fuel the SMT scheduling engine. The results shown in Table 3 are presented for an aggressive 8-way issue machine. It is conceivable that if the issue bandwidth is reduced, then the impact of disabling control flow speculation will be less severe. Table 4 summarizes the results for a 4-way issue machine. As expected, while the relative performance degradations are lower in this case, the magnitude of the absolute losses is still very significant, even for relatively highly multi-threaded workloads.

**Table 3: DISPEX Performance**

| # Contexts | 1 | 2 | 3 | 4 | 8 |
|---|---|---|---|---|---|
| Baseline IPC | 3.12 | 4.50 | 4.76 | 5.03 | 5.16 |
| DISPEX IPC | 0.88 | 1.42 | 1.70 | 2.19 | 3.02 |
| Throughput IPC Loss | 71.79% | 68.44% | 64.29% | 56.46% | 41.47% |
| Baseline Fairness | | 0.72 | 0.55 | 0.41 | 0.20 |
| DISPEX Fairness | | 0.23 | 0.19 | 0.18 | 0.11 |
| Fairness Loss | | 68.06% | 65.45% | 56.1% | 45% |

**Table 4: IPC with Reduced Issue Bandwidth (4)**

| | 2 Contexts | 3 Contexts | 4 Contexts |
|---|---|---|---|
| Baseline IPC | 2.72 | 2.81 | 2.80 |
| DISPEX IPC | 1.3 | 1.54 | 1.95 |
| IPC Loss | 52.21% | 45.2% | 30.36% |

When the control flow speculation is disabled and a thread is waiting for a branch to resolve, it is unable to make further progress. To reduce this effect, threads can be allowed to speculatively fetch, but not execute, instructions. These instructions remain in the instruction fetch queue (IFQ) until it can be determined if they are situated on the correct path or need to be flushed from the IFQ. If the prediction is correct, the thread can resume execution and avoid fetch-to-rename delays for the instructions that were speculatively fetched. On a misprediction, the IFQ is flushed to recover precise processor state; this penalty is less significant than that encountered in the course of the branch misprediction handling with full-fledged control flow speculation in place, since none of these instructions had been issued.

This variation can be implemented in the following way. When a conditional branch instruction is encountered, further dispatches (after the branch) are prevented by setting a *Dispatch_Disable* bit for that thread and allowing successive instructions to remain in the IFQ. Once the outcome of the branch is known, it is compared against the prediction – if prediction is incorrect the front-end is flushed and the *Dispatch_Disable* bit is reset to allow the thread to resume dispatch. Stalls due to waiting for branch resolution can be further reduced by adding DEdicated Branch Resolution Logic (DEBRL) for each hardware context that attempts to resolve the branch instruction that set the *Dispatch_Disable* bit. Whenever a branch undergoes register renaming, its source physical registers can be compared immediately (if they are ready, otherwise the branch is processed in the usual fashion) and the branch outcome can be provided without waiting for the actual execution. In this manner, the *Dispatch_Disable* bits can be reset potentially much earlier. This mechanism, called DISPEX+SF (DISPEX + Speculative Fetch) is an improvement compared to DISPEX (Table 3) and the benefits come at the cost of requiring a branch predictor. Full-blown branch misprediction recovery hardware is still not needed. The primary advantage of this technique is in the more efficient use of the shared dynamic scheduling logic, as only the non-speculative instructions (the ones guaranteed to be on the correct execution path) are considered for scheduling. The impact of these mechanisms on the performance is shown in Table 5. While an improvement over the previous schemes, the overall performance losses are still significant compared to the model with speculative execution.

Another potential improvement in performance can be achieved by modifying the existing fetching scheme (such as ICOUNT) by giving the highest fetching priority to the threads known to execute along the non-speculative paths. This idea was also explored in [18]. This can be accomplished by allowing ICOUNT to disregard any threads with a *Fetch_Disable* or *Dispatch_Disable* bit set and selecting from the remaining threads (when only one non-speculative thread exists it is given full fetch bandwidth). When all threads fetch based on prior speculations, the unmodified ICOUNT is used. Additional performance can be harvested by enhancing DEBRL to attempt branch resolution every cycle (Aggressive DEBRL, or ADEBRL). This should improve performance in all cases since the duration of the stall periods can only shorten. As expected, implementing a speculation aware fetch policy and aggressive DEBRL shrunk the performance gap compared to the fully speculative

processor, but the absolute performance losses are still significant – almost 25% on the average across the simulated mixes for 4-threaded workloads. These results are shown in Table 6.

In conclusion, our evaluations showed that branch speculation is still a very important performance-enhancing technique for SMT machine, even with a fairly large number of threads.

**Table 5: DISPEX+SF Performance**

| # Contexts | 1 | 2 | 3 | 4 | 8 |
|---|---|---|---|---|---|
| Baseline IPC | 3.12 | 4.50 | 4.76 | 5.03 | 5.16 |
| DISPEX+SF IPC | 1.51 | 2.46 | 3.08 | 3.59 | 4.17 |
| IPC Loss | 51.6% | 45.33% | 35.29% | 28.63% | 19.19% |
| Baseline Fairness | | 0.72 | 0.55 | 0.41 | 0.20 |
| DISPEX+SF Fairness | | 0.39 | 0.33 | 0.29 | 0.13 |
| Fairness Loss | | 45.83% | 40% | 29.27% | 35% |

**Table 6: ADISPEX+SF Performance**

| | 2 Contexts | 3 Contexts | 4 Contexts |
|---|---|---|---|
| Baseline IPC | 4.50 | 4.76 | 5.03 |
| ADISPEX+SF IPC | 2.56 | 3.15 | 3.79 |
| IPC Loss | 43.11% | 33.82% | 24.65% |
| Baseline Fairness | 0.72 | 0.55 | 0.41 |
| ADISPEX+SF Fairness | 0.41 | 0.33 | 0.31 |
| Fairness Loss | 43.06% | 40% | 24.39% |

## 5. Impact of Speculative Scheduling on SMT

This section quantifies the impact of using non-speculative scheduling on SMT and requiring all load-dependent instructions to wait for the load to be serviced before they can be scheduled. Although most loads hit into the L1 cache [19] the variable latency of load instructions can result in cache misses that require the replay of instructions that were scheduled in anticipation of the load – wasting resources that can be used for processing other instructions. Non-speculative scheduling allows the load-hit predictor to be eliminated, requiring all load dependent instructions to wait until their source registers are ready instead of being speculatively scheduled to pick up the load's result just-in-time from the bypass network; avoiding issue-to-execute delay. This causes a pipeline bubble to occur between the load and all of its dependent instructions which will degrade performance. The exact magnitude of this performance loss depends on the depth of the issue-to-execute portion of the pipeline and the number of simultaneous threads. A less complex pipeline with small issue-to-execute depth is expected to be less dependent on speculative scheduling and highly multi-threaded workloads should also experience smaller performance degradations due to the ability to cover the gaps in the execution schedule with the supply of independent instructions from multiple streams.

Non-speculative scheduling can be implemented by requiring all instructions to check for the availability of their source operands from the physical register file rather than using load-hit predictor. This avoids scheduling replays [9], but also disallows back-to-back execution of loads and load dependent instructions. In our experiments, we vary the depth of schedule-to-execute portion of the pipeline between 1, 2, and 3 cycles. Unless otherwise indicated, a 3 cycle delay is used.

First, we quantify the impact of non-speculative scheduling in the framework of a single-threaded superscalar machine. Figure 1 shows the impact of removing speculative scheduling with various issue-to-execute delays compared to the baseline single-threaded superscalar. With the issue-to-execute pipeline depth of 3 cycles, performance losses range from 8.63% (*mgrid*) to 70.14% (*wupwise*) with an average of 37.90%. These losses decrease as the issue-to-execute latency is reduced, as expected – demonstrating that speculative scheduling is less important with a simpler pipeline. With a 2-cycle issue-to-execute pipeline, the performance losses ranged from 5.24% (*mgrid*) to 60.18% (*wupwise*) with an average of 26.22%. When the depth of the issue-to-execute pipeline is reduced down to a single cycle, *eon* showed an improvement of 1.94% with *wupwise* showing the most degradation at 40.28% and an overall average degradation of 10.3%.

Table 9 summarizes the impact of using non-speculative scheduling on SMT machine with various numbers of threads and different issue-to-execute pipelines. In general, the performance losses are small for shallow pipelines and significant for deep pipelines for any number of threads. For example, the execution of 4-threaded workloads on a processor with 3-stage issue-to-execute pipeline can completely cover the lack of speculative scheduling (approaching the performance of the machine with speculative scheduling within 0.4%). In all other cases that we examined, either moderate or significant performance losses are encountered. Figure 2 shows the per-mix results for the situation with a single cycle issue-to-execute pipeline. As seen from the figure, the performance losses are quite small for all simulated workloads. Moreover, some mixes showed improvement compared to their baseline equivalent. These are Mix 2-1 (perlbmk, eon), Mix 3-5 (mgrid, fma3d, art), Mix 4-8 (parser,

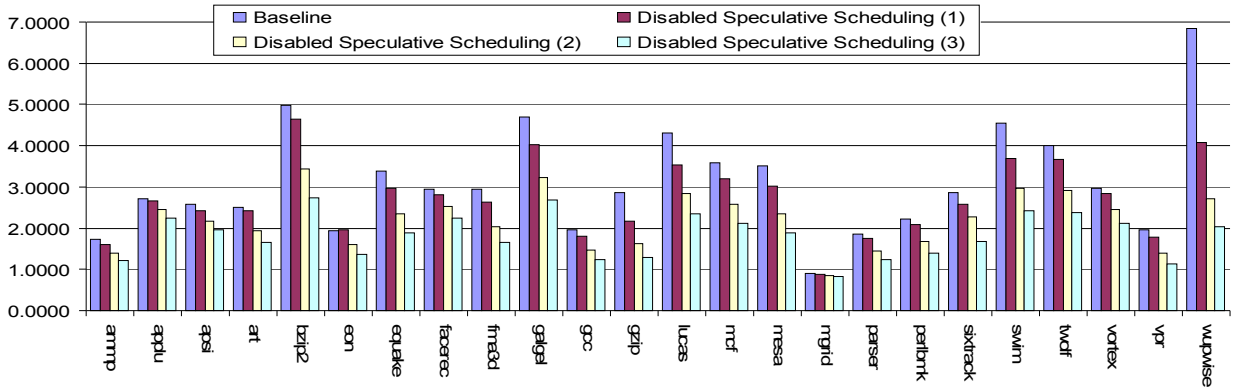


**Figure 1: Single-Threaded IPC Performance with Non-speculative Scheduling**

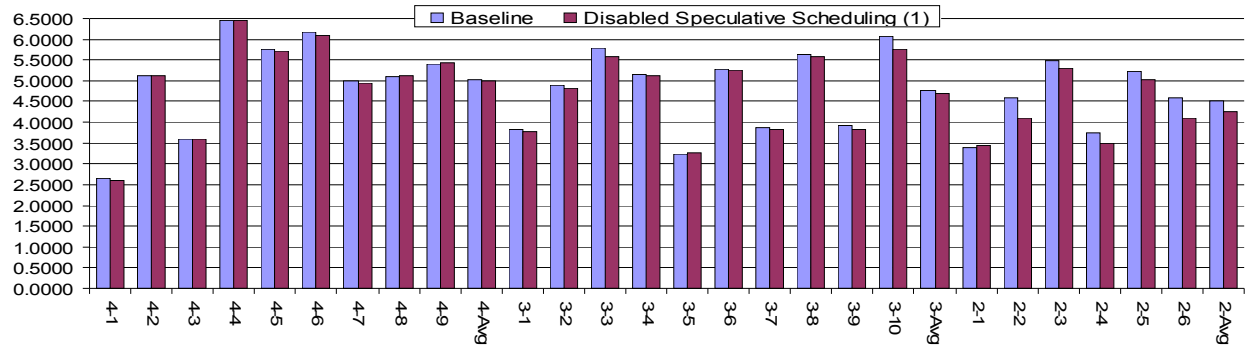


**Figure 2: SMT Performance with Non-Speculative Scheduling: 1-Cycle Issue-to-Execute Latency**

**Table 9: Performance with Disabled Speculative Scheduling**

| # Contexts | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Baseline IPC | 3.12 | 4.50 | 4.76 | 5.03 |
| 3 Cycle Delay - IPC | 1.82 | 2.99 | 3.55 | 3.91 |
| 2 Cycle Delay - IPC | 2.2 | 3.57 | 4.14 | 4.54 |
| 1 Cycle Delay - IPC | 2.72 | 4.25 | 4.68 | 5.01 |
| 3 Cycle Delay - IPC Loss | 41.67% | 33.56% | 25.42% | 22.27% |
| 2 Cycle Delay - IPC Loss | 29.49% | 20.67% | 13.03% | 9.74% |
| 1 Cycle Delay - IPC Loss | 12.82% | 5.56% | 1.68% | 0.40% |
| Baseline Fairness | | 0.72 | 0.55 | 0.41 |
| 3 Cycle Delay - Fairness | | 0.49 | 0.41 | 0.32 |
| 2 Cycle Delay - Fairness | | 0.58 | 0.48 | 0.37 |
| 1 Cycle Delay - Fairness | | 0.68 | 0.54 | 0.41 |
| 3 Cycle Delay - Fairness Loss | | 31.94% | 25.45% | 21.95% |
| 2 Cycle Delay - Fairness Loss | | 19.44% | 12.73% | 9.76% |
| 1 Cycle Delay - Fairness Loss | | 5.56% | 1.82% | 0% |

**Table 10: Performance Comparison of Atomic and Pipelined Schedulers**

| # Contexts | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Atomic Scheduler IPC | 2.99 | 4.24 | 4.63 | 4.92 |
| Pipelined Scheduler IPC | 2.64 | 4.00 | 4.41 | 4.75 |
| IPC Loss | 11.63% | 5.66% | 4.75% | 3.46% |
| Atomic Scheduler Fairness | | 0.68 | 0.54 | 0.40 |
| Pipelined Scheduler Fairness | | 0.64 | 0.51 | 0.39 |
| Fairness Loss | | 5.88% | 5.56% | 2.5% |

perlbmk, equake, lucas) and Mix 4-9 (eon, apsi, facerec, swim) shown in Figure 2.

## 6. Impact of Pipelining the Instruction Scheduling Logic on SMT

In this section we examine the impact of pipelining the scheduling logic (wakeup and selection) over two cycles – on an SMT processor. Without back-to-back execution, dependent instructions can not be issued in successive cycles and performance degrades as a pipeline bubble is introduced between those instructions. As it becomes increasingly more difficult to implement the tight scheduling loop within a single cycle, it is important to examine the impact of pipelining this logic over two cycles on SMT machine and find out if there is sufficient TLP to cover up the resulting pipeline bubbles. Unless otherwise specified, we use position-based selection in these experiments (the instructions are selected starting from one end of the issue queue).

Table 10 shows the performance of both the atomic (wakeup and select are implemented in a single cycle) and pipelined (wakeup and selection are two separate cycles) schedulers. As before, the results are presented as averages across all benchmarks or all simulated mixes for multithreaded workloads. For single-threaded workloads the performance losses resulting from pipelining the scheduling logic are almost 12% - this is in line with the previously reported estimates [2]. However, as the degree of multithreading is increased, much smaller losses are encountered. For example, in a 4-way SMT, pipelined implementation of the scheduler results in only 3.5% degradation in throughput and 2.5% degradation in fairness. This is because the bubbles created in the execution schedule of one thread due to the lack of back-

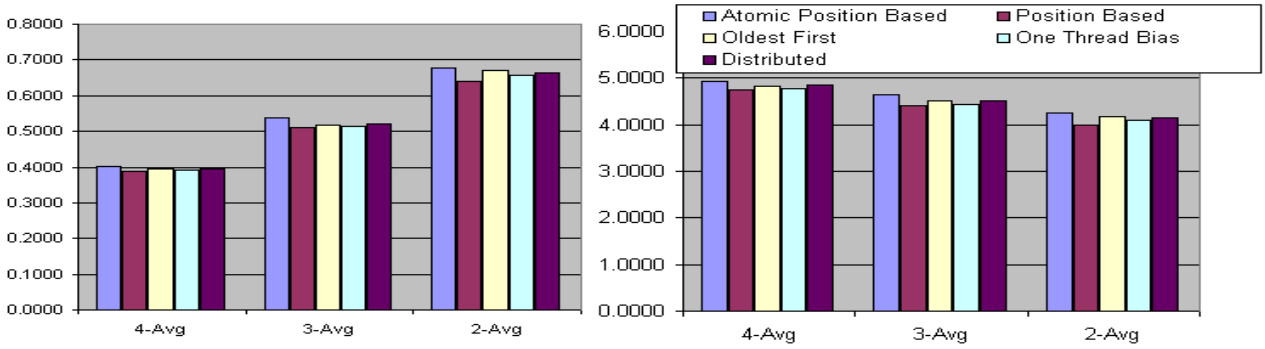


**Figure 3: Pipelined Scheduler Performance for 4-threaded Workloads**

to-back execution are effectively filled by instructions from other threads. In the next subsection, we examine alternative selection schemes in an effort to further reduce the impact of pipelining the scheduling logic on SMT.

### 6.1 Alternative Selection Schemes

In order to further reduce the negative impact of pipelining the scheduling logic, we examine the selection schemes that take the SMT environment into consideration and make decisions based on the knowledge of how many hardware contexts exist. Both schemes use a counter to keep track of how many instructions from a given thread are waiting in the IQ as well as gatekeeper logic that can divide the issue-width among the hardware contexts with fine granularity. If there are fewer instructions in the IQ than the issue-width then the gatekeeper issues all instructions and disregards the scheduling logic. The specific schemes are:

- **One Thread Bias Selection** favors a different thread, using round robin, each cycle. The gatekeeper is told to allow all the instructions from that thread, if possible, and give any remaining issue-width to the next thread (and so on).
- **Distributed Selection** attempts to evenly select from each of the threads in order to execute a diverse mix of instructions. As long as some issue-width remains unused, the gatekeeper is told to issue 1 instruction from each thread that has schedulable instructions remaining. When more threads have an instruction available than remaining issue-width, the decision is made by round robin.

The results of these selection schemes, as well as the results of the oldest-first selection, are shown in the following graphs. In all cases, the performance losses for all of the simulated workloads were no more than 4.1% (5.56% fairness loss) and excluding Biased Selection (the worst performer of the alternative schedulers) the average performance losses did not exceed 3.07% (2.94% fairness loss). Oldest First worked best for 2-threaded workloads and Distributed for 3 and 4-threaded workloads – this is expected as the Distributed scheme is designed to provide a balanced mix of instructions from all threads. For 4-threaded workloads, the distributed scheme comes within 1.6% of the performance of the atomic scheduler. The results are shown as an average across all mixes for 2, 3, and 4-threaded workloads in Figure 3. The figures clearly demonstrate how closely these schemes approximate the baseline machine and also show the consistency of the results.

Even without back-to-back execution the combination of TLP and improved selection logic approximate an atomic design with position-based selection. The summary of all the results from this section is shown in Table 11.

**Table 11: Performance of Various Pipelined Schedulers**

| # Contexts | 2 | 3 | 4 |
|---|---|---|---|
| Atomic Position Based IPC | 4.24 | 4.63 | 4.92 |
| Oldest First IPC | 4.18 | 4.50 | 4.84 |
| One Thread Bias IPC | 4.11 | 4.44 | 4.78 |
| Distributed IPC | 4.14 | 4.50 | 4.85 |
| Oldest First IPC Loss | 1.42% | 2.81% | 1.63% |
| One Thread Bias IPC Loss | 3.07% | 4.1% | 2.85% |
| Distributed IPC Loss | 2.36% | 2.81% | 1.42% |
| Atomic Position Based Fairness | 0.68 | 0.54 | 0.40 |
| Oldest First Fairness | 0.67 | 0.52 | 0.39 |
| One Thread Bias Fairness | 0.66 | 0.51 | 0.39 |
| Distributed Fairness | 0.66 | 0.52 | 0.40 |
| Oldest First Fairness Loss | 1.47% | 3.7% | 2.5% |
| One Thread Bias Fairness Loss | 2.94% | 5.56% | 2.5% |
| Distributed Fairness Loss | 2.94% | 3.7% | 0% |

## 7. Concluding Remarks

Aggressive speculative execution and instruction scheduling techniques require significant implementation complexity in order to improve performance. In this paper, we performed a comprehensive and systematic study of the performance impact of speculative execution and aggressive instruction scheduling on SMT machine. Specifically, we attempted to understand whether the presence of thread-level parallelism on SMT processors still makes it necessary to: 1) perform speculative execution based on branch prediction, 2) schedule the load-dependent instructions speculatively based on the prediction of cache hit or miss, and 3) implement the scheduling logic as an atomic operation to support back-to-back execution within each thread. Our results showed that the answer to the first question is a definite "yes", the answer to the second question strongly depends on the

number of threads and the depth of the issue-to-execute pipeline, and the answer to the third question is most cases "no".

Specifically, the performance losses encountered when speculation on branches is not allowed are very high even with highly-muiltithreaded workloads due to the inability to provide a "good" mix of instructions from multiple threads to the dynamic scheduler. When speculative scheduling is not used, an SMT running 4-threaded workloads with an issue-to-execute depth of one stage has performance that comes within 0.4% of the baseline machine. As the issue-to-execute depth increases, significantly higher performance losses are encountered. Finally, we observed that pipelined implementation of the scheduling logic can be an attractive solution for SMT, especially for larger number of threads. For example, the performance of a 4-way SMT machine with pipelined schedulers and distributed selection comes to within 1.6% of the performance of the idealistic machine with atomic scheduling logic.

## 8. Acknowledgements

Jason Loew is partially supported by the Clifford D. Clark Graduate Fellowship.

## 9. References

[1] Berger, D., Austin, T. "The SimpleScalar tool set: Version 2.0", Technical Report, Department of Computer Science , University of Wisconsin-Madison, June 1997.

[2] Brown, M., Stark, J., Patt, Y. "Select-Free Instruction Scheduling Logic", 34th International Symposium on Microarchitecture, December 2001.

[3] Butler, M., Patt, Y. "An Investigation of the Performance of Various Dynamic Scheduling Techniques", 25th International Symposium on Microarchitecture, 1992, p.1-9.

[4] Chrysos, G.Z., Emer, J.S. "Memory Dependence Prediction using Store Sets", 25th International Symposium on Microarchitecture, June 1998.

[5] Gwennap, L. "New Algorithm Improves Branch Prediction", Microprocessor Reports, Vol 9, March 1995, p. 17-21.

[6] Henning, J.L. "SPEC CPU2000: Measuring CPU Performance in the New Millennium", IEEE Computer, July 2000, p. 28-35.

[7] Jiménez, D.A. "Reconsidering Complex Branch Predictors", 9th International Symposium on High Performance Computer Architecture, 2003.

[8] Kim, Ilhyun., Lipasti, M.H. "Macro-op Scheduling: Relaxing Scheduling Loop Constraints", 36th International Symposium on Microarchitecture, 2003, p. 277.

[9] Kim, Ilhyun., Lipasti, M.H. "Understanding Scheduling Replay Schemes", 10th International Symposium on High Performance Computer Architecture, 2004, p. 198.

[10] Luo, K., Gummaraju, J., Franklin, M. "Balancing Throughput and Fairness in SMT Processors", International Symposium on Performance Analysis of Systems and Software, January 2001, p. 161-171.

[11] McFarling, S. "Combining Branch Predictors", DEC Western Research Laboratory Technical Note TN-36, June 1993.

[12] Moreshet, T., Bahar, R.I. "Complexity-Effective Issue Queue Design Under Load-Hit Speculation", Workshop on Complexity-Effective Design, May 2002.

[13] Pan, S., So, K., Rahmeh, J.T. "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation", 5th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1992, p. 76- 84.

[14] Ramsay, M., Feucht, C., Lipasti, M.H. "Exploring Efficient SMT Branch Predictor Design", ISCA Workshop on Complexity Design, June 2003.

[15] Sharkey, J.J., Ponomarev, D., Ghose, K. "M-SIM: A Flexible, Multithreaded Architectural Simulation Environment", Technical Report CS-TR-05-DP01, Department of Computer Science, State University of New York at Binghamton, October 2005.

[16] Smith, J.E. "A Study of Branch Prediction Strategies", Proceedings of ISCA-8, 1981.

[17] Stark, J., Brown, M.D., Patt, Y.N. "On Pipelining Dynamic Instruction Scheduling Logic", 33rd International Symposium on Microarchitecture, 2000.

[18] Swanson, S., McDowell, L.K., Swift, M.M., Eggers, S.J., Levy, H.M. "An Evaluation of Speculative Instruction Execution on Simultaneous Multithreaded Processors.", ACM Transactions on Computer Systems, Vol 21, Issue 3, August 2003, p. 314-340.

[19] Tullsen, D.M., Brown, J.A. "Handling Long-Latency Loads in a Simultaneous Multithreading Processors", 34th International Symposium on Microarchitecture, December 2001.

[20] Tullsen, D.M., Eggers, S.J., Emer, J.S., Levy, H.M., Lo, J.L., Stamm, R.L. "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor", 23rd Annual Symposium on Computer Architecture, May 1996.

[21] Tullsen, D.M., Eggers, S., Levy, H. "Simultaneous Multithreading: Maximizing On-Chip Parallelism", 22nd Annual International Symposium on Computer Architecture, June 2005.

[22] Yoaz, A., Erez, Mattan., Ronen, R., Jourdan, S. "Speculation Techniques for Improving Load Related Instruction Scheduling", 26th International Symposium on Computer Architecture, May 19.