In Proceedings of the 34th International Symposium on Microarchitecture, December, 2001

Handling Long-latency Loads in a Simultaneous Multithreading Processor

Dean M. Tullsen Jeffery A. Brown

Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114
{tullsen,jbrown}@cs.ucsd.edu

Abstract

Simultaneous multithreading architectures have been defined previously with fully shared execution resources. When one thread in such an architecture experiences a very long-latency operation, such as a load miss, the thread will eventually stall, potentially holding resources which other threads could be using to make forward progress.

This paper shows that in many cases it is better to free the resources associated with a stalled thread rather than keep that thread ready to immediately begin execution upon return of the loaded data. Several possible architectures are examined, and some simple solutions are shown to be very effective, achieving speedups close to 6.0 in some cases, and averaging 15% speedup with four threads and over 100% speedup with two threads running. Response times are cut in half for several workloads in open system experiments.

1 Introduction

Simultaneous multithreading (SMT) [18, 17, 20, 7] is an architectural technique that allows a processor to issue instructions from multiple hardware contexts, or threads, to the functional units of a superscalar processor in the same cycle. It increases instruction-level parallelism available to the architecture by allowing the processor to exploit the natural parallelism between threads each cycle.

Simultaneous multithreading outperforms previous models of hardware multithreading primarily because it hides short latencies (which can often dominate performance on a uniprocessor) much more effectively. For example, neither fine-grain multithreaded architectures [2, 8], which context switch every cycle, nor coarse-grain multithreaded architectures [1, 12], which context switch only on long-latency operations, can hide the latency of a single-cycle integer add if there is not sufficient parallelism in the same thread.

What has not been shown previously is that an SMT processor does not necessarily handle very long-latency operations as well as other models of multithreading. SMT typically benefits from giving threads complete access to all resources every cycle, but when a thread occupies resources without making progress, it can impede the progress of other threads. In a coarse-grain multithreaded architecture, for example, a stalled thread is completely evicted from the processor on a context switch; however, with SMT a stalled thread continues to hold instruction queue or reservation station space, and can even continue fetching instructions into the machine while it is stalled.

This research demonstrates that an SMT processor can be throttled by a single thread with poor cache behavior; however, by identifying threads that become stalled, and limiting their use of machine resources, this problem can be eliminated. This provides not only significantly higher overall throughput, but also more predictable throughput, as threads with good cache behavior are much more insulated from co-scheduled threads with poor cache behavior.

In many cases it is better to free the resources associated with a stalled thread rather than keep that thread ready to immediately begin execution upon return of the loaded data. Several possible architectures are examined, and some simple solutions are found to be very effective, achieving speedups close to 6.0 in some cases, and averaging 15% speedup with four threads and over 100% speedup with two threads running.

This paper is organized as follows. Section 2 demonstrates the long-latency load problem. Section 3 discusses previous and related work. Section 4 describes the measurement methodology and Section 5 discusses the metrics used in this study. Section 6 presents mechanisms for identifying threads stalled waiting for long loads, and for freeing resources once such loads are identified. Section 7 discusses alternate mechanisms for freeing those resources. Section 8 presents results for a more conclusive set of response time experiments using the presented solutions, and Section 9 demonstrates that the load problem exists across a variety of architectural parameters. We conclude with Section 10.

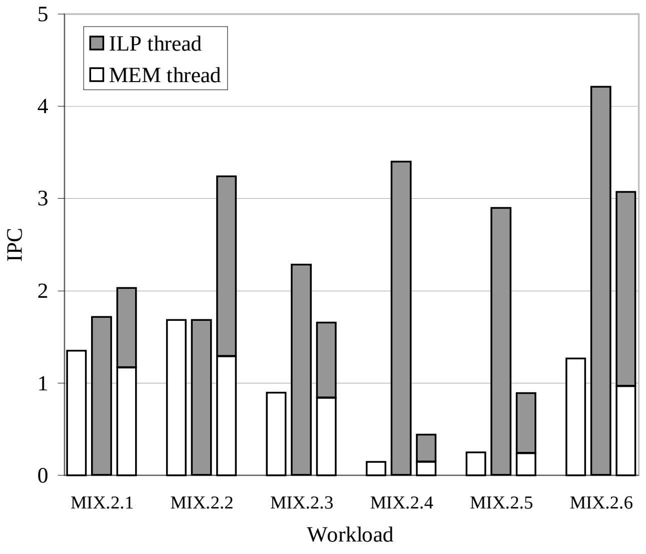

Figure 1. The performance of several two-thread mixes of memory-bound and ILP-bound applications. The stacked bars represent two-thread runs, the single bars represent the single-thread runs for the same two benchmarks.

2 The Impact of Long-latency Loads

This section demonstrates the problem of long-latency loads with a simple experiment, shown in Figure 1. For six combinations of two threads (the actual workloads and experimental configuration are described in Section 4), the figure shows three results: the IPC of each of the two threads running alone, and of the two threads running together on the SMT processor. In each case the light bars represent memory-intensive benchmarks, and the gray bars represents applications with good cache behavior.

These results show that one thread with poor cache performance can become a significant inhibitor to another thread with good cache behavior. There are two factors that allow an application with poor cache locality to cripple co-scheduled applications. First, an application that regularly sweeps through the shared cache will evict data from the other applications, degrading their cache hit rates. Second, the memory-bound application can hold and/or use critical execution resources while it is not making progress due to long-latency memory operations, degrading every thread's performance. This research focuses on the latter problem.

Few applications contain sufficient parallelism to hide long memory operations (e.g., more than a dozen cycles). While multithreading allows other threads to hide that latency, if the stalled thread fills the instruction queue with waiting instructions, it shrinks the window available for the other threads to find instructions to issue. Thus, when parallelism is most needed (when one or more threads are no longer contributing to the instruction flow), fewer resources are available to expose that parallelism.

This is most clearly demonstrated for the instruction queues by the MIX.2.5 workload, where the integer queue is on average 97% occupied when at least one L2 miss is outstanding, but only 62% occupied at other times. Other resources that are potentially held or used by a thread stalled waiting for a long memory operation are renaming registers and fetch/decode bandwidth. The rest of the paper will demonstrate that contention for shared resources is by far the dominant factor causing the poor performance shown in Figure 1.

3 Related Work

Simultaneous multithreading [18, 17, 20, 7] is an architectural technique that allows a processor to issue instructions from multiple hardware contexts, or threads, to the functional units of a superscalar architecture each cycle. This paper builds upon the SMT architecture presented in [17]. Previous SMT research has not exposed the problem (or solutions) examined in this paper. One important reason for that has been the inability of pre-2000 instantiations of the SPEC benchmarks to put significant pressure on a reasonable cache hierarchy.

Less aggressive models of multithreading are less prone to such problems. Coarse-grain multithreading [1, 12] is aimed *only* at the long-latency load problem, and makes no attempt to address any other machine latency. Because coarse-grain architectures allow only one thread to have access to execution resources at any time, they alway flush stalled threads completely from the machine. Fine-grain multithreading [2, 8] could potentially have shared scheduling resources which exhibit this problem, depending on the architecture. However, these architectures (e.g., the Cray/Tera MTA [2]) have traditionally been coupled with in-order execution, where scheduling windows only need to keep a few instructions per thread visible.

We ignore the latency of synchronization operations (the other source of long and non-deterministic latencies) in this paper. Tullsen, et al.[19] have shown the advantage of a synchronization primitive which both blocks and flushes a thread from the queue when it fails to acquire a lock; however, the performance implications of not flushing were not investigated, and that paper gives no indication that a similar technique is necessary for loads.

Previous work on the interaction of SMT processors and the cache hierarchy have focused on cache size and organization (Nemirovsky and Yamamoto [11]), cache bandwidth limitations (Hily and Seznec [6]), or cache partitioning [18].

Cache prefetching [3, 10] attacks the long-latency load problem in a different way, seeking to eliminate the latency itself. Recent work in prefetching targets multithreaded processors specifically, using idle hardware contexts to initiate prefetching. These include Collins, et al. [4], Luk [9], and Zilles and Sohi [21].

Benchmark	Input	Fast Forward
Memory-Intensive		
ammp	ref	1.7 billion
applu	ref	.7
art	c756hel.in (ref)	.2
mcf	ref	1.3
swim	ref	.5
twolf	ref	1
ILP-Intensive		
apsi	ref	.8 billion
eon	cook (ref)	1
fma	ref	.1
gcc	integrate.i (ref)	.5
gzip	log (ref)	.1
vortex	ref	.5

Table 1. The benchmarks used in this study, along with the data set and the number of instructions emulated before beginning measured simulation.

Name	Applications
ILP.2.1	apsi, eon
ILP.2.2	fma3d, gcc
ILP.2.3	gzip, vortex
ILP.4.1	apsi, eon, fma3d, gcc
ILP.4.2	apsi, eon, gzip, vortex
ILP.4.3	fma3d, gcc, gzip, vortex
MEM.2.1	applu, ammp
MEM.2.2	art, mcf
MEM.2.3	swim, twolf
MEM.4.1	ammp, applu, art, mcf
MEM.4.2	art, mcf, swim, twolf
MEM.4.3	ammp, applu, swim, twolf
MIX.2.1	applu, vortex
MIX.2.2	art, gzip
MIX.2.3	swim, gcc
MIX.2.4	ammp, fma3d
MIX.2.5	mcf, eon
MIX.2.6	twolf, apsi
MIX.4.1	ammp, applu, apsi, eon
MIX.4.2	art, mcf, fma3d, gcc
MIX.4.3	swim, twolf, gzip, vortex

Table 2. The multithreaded workloads used.

4 Methodology

Table 1 summarizes the benchmarks used in our simulations. All benchmarks are taken from the SPEC2000 suite and use the reference data sets. Six are memory-intensive applications which in our system experience between 0.02 and 0.12 L2 cache misses per instruction, on average, over the simulated portion of the code. The other six benchmarks are taken from the remainder of the suite and have lower miss rates and higher inherent ILP. Table 2 lists the multithreaded workloads used in our simulations. All of the simulations in this paper either contain threads all from the first group (the MEM workloads in Table 2), or all from the second group (ILP), or an equal mix from each (MIX). Most of the paper focuses on the MIX results; however, the other results are shown to demonstrate the universality of the problem.

Parameter	Value
Fetch width	8 instructions per cycle
Fetch policy	ICOUNT.2.8 [17]
Pipeline depth	8 stages
Min branch misprediction penalty	6 cycles
Branch predictor	2K gshare
Branch Target Buffer	256 entry, 4-way associative
Active List Entries	256 per thread
Functional Units	6 Integer (4 also load/store), 3 FP
Instruction Queues	64 entries (32 int, 32 fp)
Registers For Renaming	100 Int, 100 FP
Inst Cache	64KB, 2-way, 64-byte lines
Data Cache	64KB, 2-way, 64-byte lines
L2 Cache	512 KB, 2-way, 64-byte lines
L3 Cache	4 MB, 2-way, 64-byte lines
Latency from previous level (with no contention)	L2 10 cycles, L3 20 cycles Memory 100 cycles

Table 3. Processor configuration.

Execution is simulated on an out-of-order superscalar processor model which runs unaltered Alpha executables. The simulator is derived from SMTSIM [15], and models all typical sources of latency, including caches, branch mispredictions, TLB misses, and various resource conflicts, including renaming registers, queue entries, etc. It models both cache latencies and the effect of contention for caches and memory buses. It carefully models execution down the wrong path between branch misprediction and branch misprediction recovery.

The baseline processor configuration used for most simulations is shown in Table 3. The instruction queues for our eight-wide processor are roughly twice the size of the four-issue Alpha 21264 (15 FP and 20 integer entries) [5]. In addition, the 21264 queues cannot typically remain completely full due to the implemented queue-add mechanism, a constraint we do not model with our queues. These instruction queues, as on the 21264, remove instructions upon issue, and thus can be much smaller than, for example, a register update unit [14] which holds instructions until retirement. Section 9 also investigates larger instruction queues.

The policies of the SMT fetch unit have a significant impact on the results in this paper. Our baseline configuration uses the ICOUNT.2.8 mechanism from [17]. The ICOUNT mechanism fetches instructions from the thread or threads least represented in the pre-execute pipeline stages. This mechanism already goes a long way towards preventing a stalled thread from filling the instruction queue (Section 9 shows how much worse the load problem becomes without ICOUNT), but this paper will show that it does not completely solve the problem. In particular, if the processor is allowed to fetch from multiple threads per cycle, it becomes more likely a stalled thread (while not of the highest priority) can continue to dribble in new instructions. Our baseline fetch policy (ICOUNT.2.8) does just that, fetching eight instructions total from two threads. Section 9 also looks at fetch policies that only fetch from one thread per cycle, demonstrating that the problem of long-latency loads persists even in that scenario.

5 Metrics

This type of study represents a methodological challenge in accurately reporting performance results. In multithreaded execution, every run consists of a different mix of instructions from each thread, making IPC (instructions per cycle) a questionable metric. This problem is most problematic when we apply policies that bias execution against a particular thread or threads. It is further exacerbated when those policies bias execution against inherently low-IPC threads.

Any policy that favors high-IPC threads boosts the reported IPC by increasing the contribution from the favored threads. But this does not necessarily represent an improvement. While the IPC over a particular measurement interval might be higher, in a real system the machine would eventually have to run a workload inordinately heavy in low-IPC threads, and the artificially-generated gains would disappear.

An actual result from this paper will illustrate. The baseline processor runs two threads together, with the first thread achieving 1.29 IPC, and the second 1.95 IPC, for a combined 3.24 IPC. With a particular optimization, the machine now runs at a slightly higher IPC of 3.25, achieved by the first thread running at 1.02 IPC and the second 2.23. However, this increase in IPC was achieved by running the first thread 21% slower and the second thread 14% faster. The processor is not making better progress through the given workload, despite the higher IPC. Any reasonable measure of system-level performance (for example, average job turnaround time), would see more degradation from the 21% slowdown than gain from the 14% speedup.

This paper will use *weighted speedup*, which is derived from, but not identical to, the metric of the same name proposed by Snavely and Tullsen [13]. In that paper, a thread's IPC is derated by its single-thread IPC; this paper derates a thread's IPC by its IPC in the baseline configuration with the same mix of threads; however, the spirit of the metric is the same: to make it impossible to quote artificial speedups by simply favoring high-IPC threads. We use this modification because, (1) in this study we have a more well-defined baseline case, and (2) when two threads are running slowly together, we benefit from either running faster, regardless of how they would run in single thread mode.

Our definition of weighted speedup is as follows:

$$\text{Weighted Speedup} \;=\; \sum_{threads} \frac{IPC_{new}}{IPC_{baseline}}$$

By this formula, then, the previously cited simulation result would achieve a weighted speedup (slowdown) of 0.97. The nature of the optimizations in this paper are such that virtually all of our results report lower weighted speedup than unweighted speedup, but we feel strongly that this is a more accurate metric.

In Section 8, we also follow the lead of [13] by using open system response time experiments to assess the benefit of these optimizations in a dynamic system with jobs entering and leaving the processor over time.

6 Identifying and Handling Long-latency Loads

This section details our primary mechanisms for (1) identifying that a thread or threads are likely stalled, and (2) freeing resources associated with those threads.

Identifying stalled threads in most cases operates on two assumptions: that only loads can incur sufficient latency to require this type of drastic action, and that if a load takes long enough, it is almost certain to stall the thread. (See [19] for a study of synchronization mechanisms on SMT, which is the other potential source of long thread stalls). Note that in an out-of-order processor, the notion of a "stalled" thread is much fuzzier than in an in-order processor. In an out-of-order processor, only those instructions dependent on the load will get stuck in the instruction queue, but if the memory latency is long enough, eventually the thread will run out of instructions that are independent of the load (or the active list/reorder buffer will fill with the stalled load at the head). At that point, the thread has gone from partially stalled to fully stalled.

Freeing resources requires removing instructions from the processor. In most of our experiments we assume that the processor uses the exact same flushing mechanism that is used for a branch misprediction, which can flush part of a thread starting at a given instruction. Such a flush frees renaming registers and instruction queue entries.

We make the following assumptions in all of the experiments in this paper. First, that we always attempt to leave one thread running; we do not flush or block a thread if all others have already been flushed or blocked. Second, that any thread which has been flushed is also blocked from further fetching until the load returns from memory. Third, that the processor core receives little advance warning that a load has returned from memory. In our case, the two-cycle cache fill time allows us to possibly begin fetching one cycle before the load data is available (roughly four cycles too late to get the first instructions in place to use the returned data immediately). A mechanism that accurately predicted the return of a load, or received that information from the memory subsystem early, would allow the thread to bring the instruction stream back into the scheduling window more quickly, achieving higher performance than shown here at the cost of some complexity.

We'll examine two mechanisms for identifying long-latency loads. *Trigger on miss* assumes we get a signal from the L2 cache on each miss, and that the processor can attribute that miss to a particular thread and instruction. We

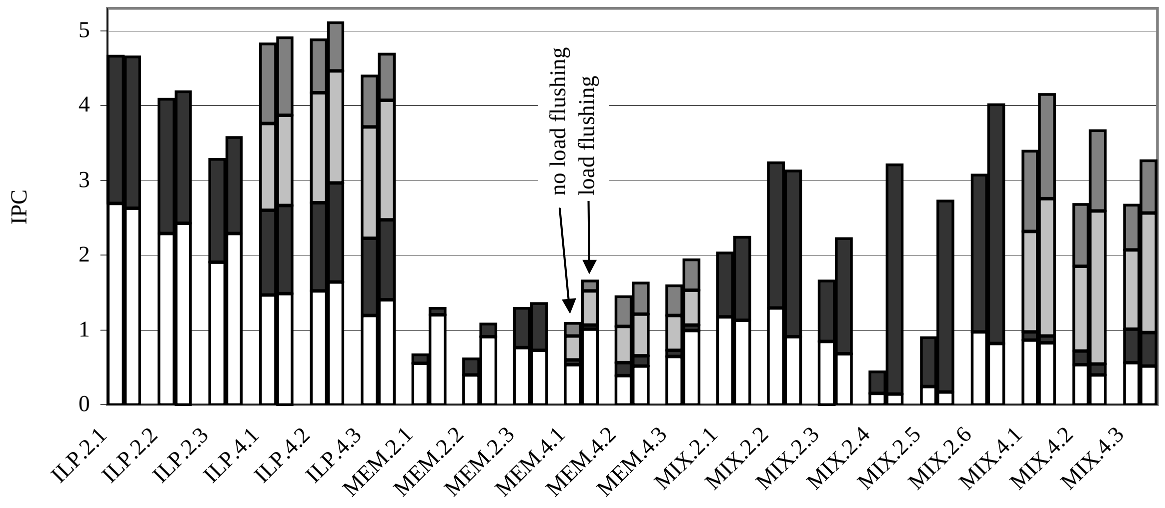

Figure 2. The instruction throughput of all workloads with a simple mechanism for flushing threads waiting for long-latency loads. The contributions by each thread to the total IPC are shown by the segmented bars. For the MIX results the memory-intensive benchmarks are those closest to the bottom of the graph. In each pair, the left bar uses no flushing, and the right bar uses the "T15" policy.

also assume that a TLB miss triggers a flush, on the assumption that most TLB misses will incur expensive accesses to fill the TLB and will often also result in cache misses after the TLB is reloaded. If the TLB miss is handled by software in the same thread context, the processor must not flush until after the miss is handled. A simpler mechanism, *trigger on delay*, just initiates action when an instruction has been in the load queue more than L cycles after the load was first executed. For most of our experiments, L is 15. That is more than the L2 hit time (10 cycles), plus a few more cycles to account for the non-determinism caused by bank conflicts and bus conflicts.

Figure 2 shows just the latter mechanism (T15 — trigger a flush after 15 cycles) compared to regular execution (no flushing) for all combinations of workloads. This graph plots instructions per cycle, for each thread, and shows that the performance gains are mostly coming from the non-memory threads being allowed to run unencumbered by the memory threads, with the memory threads suffering slightly. Because of the difficulties discussed in Section 4 with using IPC as a performance metric, all further graphs will show only the weighted speedup results; however, Figure 2 does give insight into how the speedups are achieved. This figure also shows that long-latency load flushing is effective even when the threads are uniform: all memory-bound or all ILP-bound. The average weighted speedup for the ILP workloads is 1.03 and for the MEM workloads is 1.25. Subsequent results will focus on the mixed workloads, however.

Figure 3 shows more mechanisms for identifying long-latency loads, including TM (trigger on L2 miss), T5, T15, and T25 (trigger a flush after a load becomes 5, 15, or 25 cycles old), and T15S (S for selective — only flush if some resource is exhausted, such as instruction queue entries or renaming registers). T5 flushes after L1 misses and T15 after L2 misses. T25 is an interesting data point, because an L3 miss takes at least 30 cycles; it will identify the same misses as T15, but identify them later.

The results are both clear and mixed. It is clear that flushing after loads is important, but the best method of triggering a flush varies by workload. Triggering after 15 cycles and triggering after a cache miss are consistently good. The selective flush is best in several cases, but also performs poorly in other cases. When it performs poorly, it is because a flush is often inevitable (especially since the stalled thread can still fetch instructions to fill the queue); then, being selective only delays the flush until some harm has actually been done and allows the doomed thread to utilize precious fetch bandwidth in the meantime. In other cases, being conservative about flushing (see both T15S and T25) pays off. This is not so much because it reduces the number of flushes, but because it allows more loads from the doomed thread to get into the memory subsystem before the flush. Thus, performance is best when we can find the right balance between the need to purge a memory-stalled thread from the machine, and the need to exploit memory-level parallelism within the memory-bound thread. That balance point varies among the workloads displayed here.

When there is little contention for the shared resources, flushing after loads can hinder one thread without aiding the other(s); in our simulations, we only see that in the MIX.2.2 workload.

The average weighted speedup for load flushing in this figure is over 2.0 when two threads are active, and 1.13–1.15 for four threads. The two-thread case is extreme because it is easy for a stalled thread to eventually take over the processor when we are fetching from two threads every cycle. How-

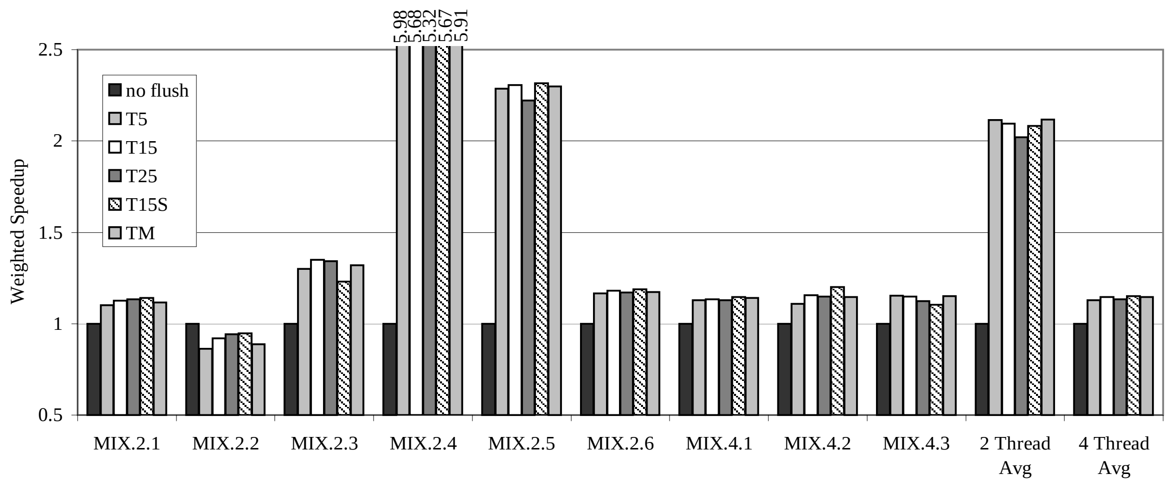

Figure 3. The weighted speedup of flushing after long loads, comparing several mechanisms for identifying long-latency loads.

ever, the four-thread case shows that even when that effect is no longer dominant, all threads still suffer from the "equal portion" of the machine which is held by a stalled thread.

Once a thread is identified as stalled and selected for flushing, the processor must choose an instruction to flush forward from; we examine several schemes. *Next* flushes beginning with the next instruction after the load. *First use* flushes at the first use of the loaded data. *After 10* and *after 20* flush beginning 10 or 20 instructions beyond the load. *Next branch* flushes at the next branch. This mechanism simplifies load flushing on processors that checkpoint only at branches. The Alpha 21264 and 21364 checkpoint all instructions, and would have more flexibility in choosing a flush point. The results presented so far have all used the flush after *first use* technique. Figure 4 shows the performance of different flush point selection techniques; the T15 load identification scheme was used for these experiments.

These results show some definite trends. When the load problem is most drastic (in the two-thread workloads, particularly MIX.2.4), it is critical to flush as close to the problem as possible, to minimize the held resources. In those cases, flushing on next, first-use, and (sometimes) first-branch all fit that bill. When the load problem is less critical, sometimes being more liberal about where to flush can actually help. However, because there is so much more to gain when the load problem is most evident, the average results are dominated by mechanisms that flush close to the load.

Further results in this paper will use the *trigger after 15 cycles* scheme to identify long loads, and will flush beginning with the *first use*. This policy will be simply denoted as T15.

The results in this section demonstrate that flushing after a long-latency load can be extremely effective in allowing non-stalled threads to make the best use of the execution resources. Flushing a thread is a fairly drastic action to take on an SMT processor, but appears warranted across a wide variety of workloads. Among the questions examined in the next section is the effectiveness of less drastic measures to solve the long-load problem.

7 Alternate Flush Mechanisms

This section investigates a wider range of mechanisms to free execution resources during long-latency loads. It seeks to answer these questions: (1) is the complexity and performance cost of flushing on long-latency loads necessary, and (2) what further benefits might be gained from more complex mechanisms?

One simpler alternative would be to only moderate fetching. That is, do not flush, but immediately stop fetching from a thread experiencing an L2 miss. This does not clear space occupied by the stalled thread, but prevents it from taking more than its share while it is not making progress. This is the *stall fetch* scheme of Figure 5.

Alternatively, we could make it more difficult for a thread to ever occupy too much of the shared queue. Certainly, a statically partitioned queue does not experience the load problem. However, that is a dear price to pay, sacrificing the most efficient use of the queue at other times, especially when not all contexts are active. A middle ground solution, however, would be a hard limit on how many instructions a single thread could have in the pre-execute portion of the pipeline (presumably this limit could be turned off when executing in single-thread mode). We experimented with several different limits, and the best performer appears as *pseudo static* in Figure 5. For that policy, no thread is allowed to fetch a new block when it has more than 20 instructions in the queue stage or earlier if we have two threads active, or more than 15 instructions if there are four threads active.

More complex mechanisms are also possible. Only slightly more complex is a hybrid of T15S and *stall fetch*. This mechanism stops fetching as soon as a long-latency load is detected, but only flushes if a resource is exhausted. Stopping fetch for the offending thread immediately increases the chances that no resource will be exhausted and no flush will

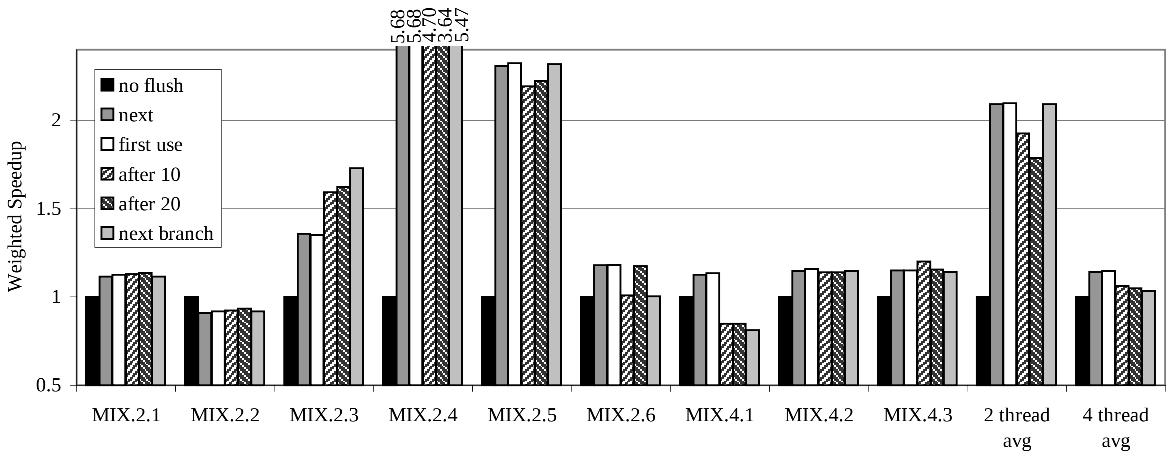

Figure 4. The weighted speedup of several mechanisms to identify the flush point after a load triggers a flush.

be necessary, if all other threads' queue pressure is light. This policy is labeled T15SF — *s*tall, then *f*lush.

The last scheme examined adds *stall buffers* to the processor. This is intended to eliminate the delay in getting instructions back into the instruction queues. Instructions that belong to a thread that is stalled, and are themselves not ready to issue, will be issued (subject to issue bandwidth constraints) to the stall buffer using the normal issue mechanism. When the load completes, instructions will be dispatched to the instruction queue (temporarily over-riding the rename-issue path), again subject to normal dispatch bandwidth. This eliminates the delay in resuming the stalled thread, allowing it to make more progress as parallelism allows. Typically, only the load-dependent instructions will go into the stall buffer, so even a small buffer can allow many independent instructions after the load to execute and avoid unnecessary squashing. Once the stall buffer fills, the thread is flushed starting with the instruction that found the buffer full.

Figure 5 shows the results. Just stalling fetch improves performance over no flushing, but falls far short of the other solutions. The pseudo-statically partitioned queue also falls short due to the inherent inefficiencies of placing artificial limits on threads' use of the queues. The stall and flush mechanism (T15SF) is a small change to our previous scheme and does show an improvement over that approach (T15). The performance of the stall buffer is disappointing. It only solves half the problem: while relieving the instruction queue, it puts more pressure on the renaming registers and increases those conflicts.

The T15 mechanism strikes a good balance between implementation complexity and performance, on a wide variety of workloads.

8 Response Time Experiments

While most of the methodological concerns with this research are eliminated with the use of weighted speedup, there are some questions that can only be answered definitively by a comprehensive model of an open system, with jobs arriving and departing. For example, one possible issue is whether even weighted speedup appropriately accounts for the fact that a continued bias against the slow threads may mean that they stay in the system longer, causing problems for more threads. In fact, we'll show that this isn't the case, a fact that is not obvious from the previous experiments.

In this experiment, we modified the simulator to allow jobs to enter the simulated system at various intervals, and run for a predetermined number of instructions. Because the runtime intervals were by necessity much less than the actual runtimes of these programs, we still fast-forwarded to an interesting portion of execution before entering the job into the system. Since the MEM threads run more slowly, we used a mix of two ILP threads to every MEM thread; this yielded a fairly even mix of jobs in the processor at any point in time. Eighteen total jobs were run, with MEM jobs run once each, and ILP jobs run twice each. In such an experiment, the only useful measure of performance is average response time (execution time), since the instruction throughput is for the most part a function of the schedule rather than the architecture. The mean response times were calculated using the geometric mean due to the wide disparity in response times for different jobs. In these simulations, all jobs execute for 300 million instructions, then exit the system. In the *light load* experiment, jobs arrive every 200 million cycles, in *medium load*, they arrive every 150 million cycles, and in *heavy load*, they arrive every 100 million cycles. For the baseline cases, there were on average 2.9, 3.4, and 4.5 jobs in the system for the light, medium, and heavy loads, respectively.

Figure 6 presents the results of the three experiments. For each experiment, the ILP and MEM thread response times are shown computed separately as well as combined. The results show dramatic decreases in response time through the use of load flushing. Surprisingly, these decreases are not restricted to the ILP threads: the MEM threads gain very significantly as well, despite being the target of bias. The gains

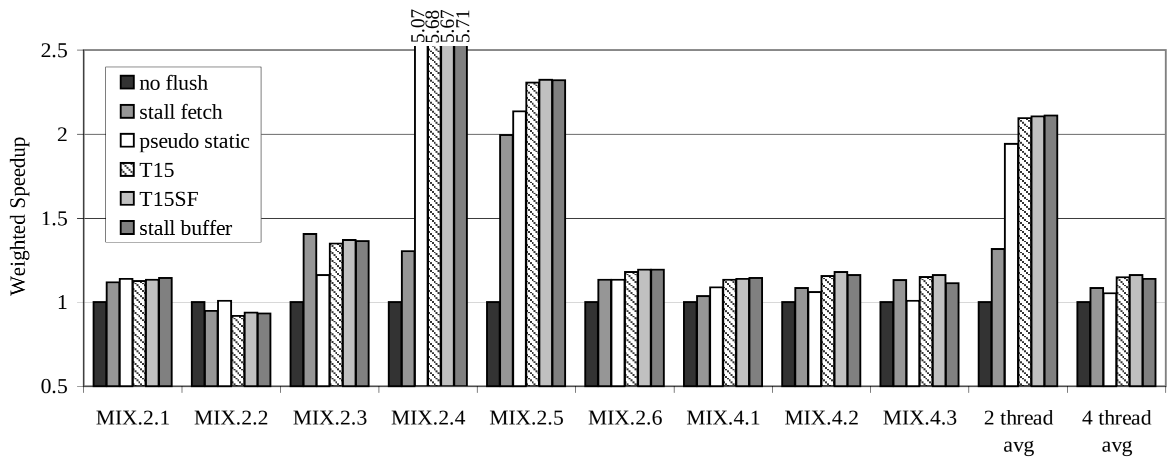

Figure 5. The performance of several alternatives to our baseline (T15) load flushing mechanism.

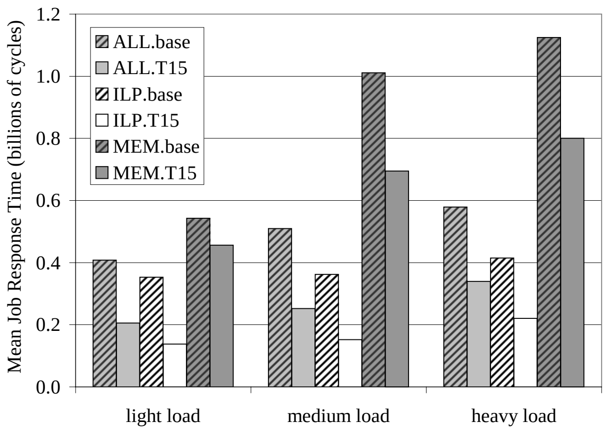

Figure 6. The mean response times of jobs in an open system experiment.

are significant with both light loads, where the average number of jobs in the system is closer to the worst-case of two threads, and with heavy loads, where the average number of jobs is much higher.

These results expose two phenomena not shown in the previous sections. First, when one thread inhibits the progress of other threads, it only causes further queueing delays as more jobs enter the system. Conversely, if a thread can accelerate a co-scheduled job's exit from the system, it gains a larger share later to accelerate its own progress. This is the source of the high speedup for the MEM threads. With the medium-load workload, load flushing reduced the average number of jobs in the system from 3.4 to 2.5, which benefited every job.

The second phenomenon which degraded the performance of the no-flushing results was the exaggeration of the two-thread problem seen in earlier results. Since this experiment saw anywhere from zero to eight threads in the system at any one time, we would hope that it would not spend too much time in the disastrous two-thread scenario. However, just the opposite took place, as the poor performance of the two-thread case made it something of a local minimum that the system constantly returned to, for some of the experiments. When more than two threads were in the system, throughput would improve, returning the system more quickly to dual execution. Similarly, the system was unlikely to move to single-thread execution if two-thread throughput was low. Thus we see that the poor dual-thread performance highlighted by previous sections will take a much larger toll on overall throughput than might be expected statistically — if it is not eliminated using the techniques outlined here.

9 Generality of the Load Problem

The benefit from flushing after long loads will vary with the parameters of the architecture. This section shows how the technique works under different assumptions about fetch policies and instruction queue size. By varying those parameters which most impact the applicability of this mechanism, this section demonstrates that these techniques solve a real problem that exists across a wide range of assumed architectures.

The effectiveness of, and necessity for, flushing after loads will necessarily vary with cache sizes and cache latency. We do not explore this space here, however, because we will be able to rely on two constants for the foreseeable future that will ensure the continued and increasing need for this technique: there will always be memory-bound applications, and memory latencies will continue to grow.

The ICOUNT fetch policy attempts to prevent a thread from ever taking more than its share of the processor. One reason that threads are able to circumvent it in these experiments is that, with a fetch policy that allows two threads to fetch concurrently, a thread not of the highest priority is still able to add instructions. Figure 7 examines the speedups achieved with various fetch policies, using the terminology from [17]. The ICOUNT.1.8 policy fetches up to eight instructions from a single thread each cycle. With that scheme a thread cannot fetch more instructions unless it is the least represented thread that is ready to fetch. The ICOUNT.2.4 scheme fetches four instructions from each of two threads for a maximum of eight. The ICOUNT.2.8 policy is the baseline

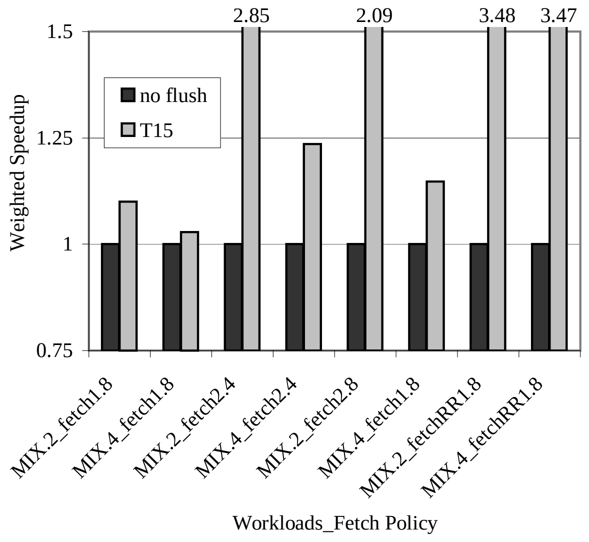

Figure 7. The weighted speedup of load flushing for different SMT fetch policies.

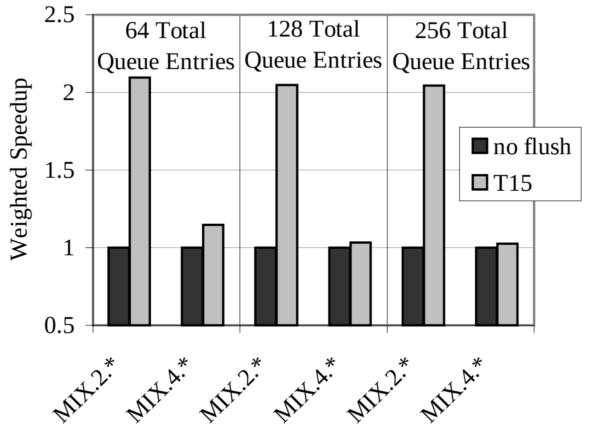

Figure 8. The weighted speedup of load flushing for different instruction queue sizes. MIX.2.* is the average for all 6 MIX.2 workloads, and MIX.4.* is the average for the MIX.4 workloads.

policy used throughout this paper. The RR.1.8 uses round-robin priority for fetch rather than ICOUNT.

Figure 7 shows that the ICOUNT.1.8 fetch policy goes a long way toward solving the problem, but it is not sufficient: there is still a significant gain for flushing, especially with two threads. This is because even if the machine is successful at preventing a thread from occupying more than an equal portion of the processor, it still loses that equal portion of the instruction window to find parallelism in other threads. Fetching from a single thread is not a panacea, anyway, because the ICOUNT.1.8 policy also has a performance cost not seen in this graph (because the speedups are normalized to different baselines). With load flushing applied, the ICOUNT.1.8 result is 9% slower than the ICOUNT.2.8 result with four threads for the MIX experiments, a result that confirms those in [17]. The ICOUNT.2.4 results show even greater gains than the ICOUNT.2.8 results. This comes from the fact that the ICOUNT.2.4 scheme gives the top two threads equal access to fetch, unlike the ICOUNT.2.8 scheme. With round-robin instruction fetching, we see to what extent the ICOUNT scheme was protecting the processor from load stalls. With round-robin fetching (the RR.1.8 results), flushing after loads is absolutely essential to good performance, regardless of the number of threads.

The size of the instruction scheduling window (in this case, the instruction queues) will also impact how easy it is for a context to monopolize the structure. Figure 8 shows the performance of load flushing for two larger queue sizes (in addition to the previous results for 64 total queue entries). As the queues become larger, the processor does become more tolerant of long-latency loads when sufficient thread parallelism exists. With fewer threads, however, it only takes the stalled thread a little longer to take over the queue, regardless of size.

Another factor that would also affect these results is the presence of other memory latency tolerance techniques, such as memory prefetching (either hardware or software). While techniques such as these are less important on a multithreaded processor, it can be expected that they will be available. In fact, much recent research is exploiting the existence of threads to create prefetching engines [4, 9, 21].

We expect this technique to co-exist efficiently with, and in some cases supplant, prefetching. No current prefetchers provide full coverage of cache misses for all important applications; so, a prefetcher could be used to boost the performance of a particular memory-intensive benchmark, while a load-flushing technique would still protect system throughput when the prefetcher fails. A hardware prefetcher for a processor that included this load-flushing mechanism would have the luxury of focusing on achieving high accuracy, because high coverage will be less important.

Some environments, however, are inappropriate for prefetching. When memory bandwidth is limited or heavily shared [16], the extra bandwidth generated by prefetching might be unacceptable, but load-flushing incurs no such cost. The extra bandwidth required for prefetching is also undesirable for low-power applications; however, the cost of re-execution after a flush may also be unacceptable, in which case stalling fetch or a static or pseudo-static partitioning of the instruction queues might become more desirable.

10 Conclusions

A thread with a high concentration of long-latency cache misses can reduce the throughput of a co-scheduled thread by as much as a factor of ten. This happens when the memory-bound thread constantly fills the instruction scheduling window with instructions that cannot be issued due to dependence on these long-latency operations. The co-scheduled thread cannot get enough instructions into the processor to expose the parallelism needed to hide the latency of the memory operation. Thus, we lose the primary advantage of multithreading.

This problem is solved by forcing threads waiting for long-latency loads to give up resources, using the same mechanism used for branch mispredictions, allowing the thread to resume fetching once the load returns from memory.

This technique achieves a 15% speedup with four threads active, and more than doubles the throughput with two threads active. Response time experiments show that under various load levels the average response time is cut by about a factor of two, including a significant reduction even for the memory-bound jobs our techniques bias against.

This paper shows that less aggressive techniques (just stalling fetch, or limiting full access to the instruction queues) can help but do not provide the speedups achieved by flushing. More aggressive techniques, such as providing dedicated buffers for holding stalled instructions, do provide some further gains, but may not justify the additional cost.

Acknowledgments

We would like to thank Srilatha Manne of Intel for helpful suggestions on this research, and the anonymous reviewers for their useful comments. This work was funded in part by NSF CAREER grant No. MIP-9701708, a Charles Lee Powell faculty fellowship, and equipment grants from Compaq Computer Corporation.

References

[1] A. Agarwal, J. Kubiatowicz, D. Kranz, B.-H. Lim, D. Yeung, G. D'Souza, and M. Parkin. Sparcle: An evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, June 1993.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. In *International Conference on Supercomputing*, pages 1–6, June 1990.

[3] T.-F. Chen and J.-L. Baer. Effective hardware-based data prefetching for high performance processors. *IEEE Transactions on Computers*, 5(44):609–623, May 1995.

[4] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *28th Annual International Symposium on Computer Architecture*, July 2001.

[5] Compaq Computer Corp., Shrewsbury, MA. *Alpha 21264 Microprocessor Hardware Reference Manual*, Feb. 2000.

[6] S. Hily and A. Seznec. Standard memory hierarchy does not fit simultaneous multithreading. In *Workshop on Multithreaded Execution Architecture and Compilation*, Jan. 1998.

[7] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *19th Annual International Symposium on Computer Architecture*, pages 136–145, May 1992.

[8] J. Laudon, A. Gupta, and M. Horowitz. Interleaving: A multithreading technique targeting multiprocessors and workstations. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 308–318, Oct. 1994.

[9] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *28th Annual International Symposium on Computer Architecture*, July 2001.

[10] T. Mowry, M. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Oct. 1992.

[11] M. Nemirovsky and W. Yamamoto. Quantitative study of data caches on a multistreamed architecture. In *Workshop on Multithreaded Execution Architecture and Compilation*, Jan. 1998.

[12] R. Saavedra-Barrera, D. Culler, and T. von Eicken. Analysis of multithreaded architectures for parallel computing. In *Second Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, July 1990.

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

[14] G. Sohi and S. Vajapeyam. Instruction issue logic for high-performance, interruptable pipelined processors. In *14th Annual International Symposium of Computer Architecture*, pages 27–31, June 1987.

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

[16] D. Tullsen and S. Eggers. Limitations of cache prefetching on a bus-based multiprocessor. In *20th Annual International Symposium on Computer Architecture*, pages 278–288, May 1993.

[17] 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 *23rd Annual International Symposium on Computer Architecture*, May 1996.

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

[19] D. Tullsen, J. Lo, S. Eggers, and H. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, Jan. 1999.

[20] W. Yamamoto and M. Nemirovsky. Increasing superscalar performance through multistreaming. In *Conference on Parallel Architectures and Compilation Techniques*, pages 49–58, June 1995.

[21] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *28th Annual International Symposium on Computer Architecture*, July 2001.