

PEEP: Exploiting Predictability of Memory Dependences in SMT Processors

Samantika Subramaniam Milos Prvulovic Gabriel H. Loh

Georgia Institute of Technology
College of Computing
{samantik,milos,loh}@cc.gatech.edu

Abstract

Simultaneous Multithreading (SMT) attempts to keep a dynamically scheduled processor's resources busy with work from multiple independent threads. Threads with long-latency stalls, however, can lead to a reduction in overall throughput because they occupy many of the critical processor resources. In this work, we first study the interaction between stalls caused by ambiguous memory dependences and SMT processing. We then propose the technique of Proactive Exclusion (PE) where the SMT fetch unit stops fetching from a thread when a memory dependence is predicted to exist. However, after the dependence has been resolved, the thread is delayed waiting for new instructions to be fetched and delivered down the front-end pipeline. So we introduce an Early Parole (EP) mechanism that exploits the predictability of dependence-resolution delays to restart fetch of an excluded thread so that the instructions reach the execution core just as the original dependence resolves. We show that combining these two techniques (PEEP) yields a 16.9% throughput improvement on a 4-way SMT processor that supports speculative memory disambiguation. These strong results indicate that a fetch policy that is cognizant of future stalls considerably improves the throughput of an SMT machine.

1. Introduction

Simultaneous Multithreading (SMT) is one of the proposed techniques for efficiently utilizing a billion transistors on a chip [7, 24]. SMT is a variation on multithreading that uses the existing resources of a multiple-issue, dynamically scheduled processor to concurrently execute instructions from different, independent threads. The advantage of an SMT processor is that it is able to exploit both instruction level parallelism and thread level parallelism, thus providing high system throughput as well as aggressive single-threaded performance. SMT machines have been designed with the view of trying to inter-mingle threads so as to maximize resource usage and improve overall throughput. However, a thread that encounters a stalling condition (e.g., a cache miss with all other independent instructions already executed) can potentially tie up many of the shared resources for the entire latency of the stall. This effectively re-

duces the number of critical resources available to the non-stalled threads, and ends up reducing overall throughput. Past work has examined the impact of branch mispredictions and cache misses on SMT performance [4, 8, 13, 14, 25]. To the best of our knowledge, no prior work has examined the effect of memory dependences on the behavior of an SMT processor.

In this paper, we first study the interaction of memory dependence prediction with a simultaneous multi-threaded processor. From these observations, we propose a new class of SMT fetch policy filters that exploit predicted memory dependences to avoid allocating resources to threads that are likely to stall on these dependences. We then use the predictability of dependence resolution latencies to mitigate thread starvation effects after dependency resolution. We also use our technique in an alternative design that does not speculatively execute loads before unresolved stores to demonstrate that our proposed technique works because of effective fetch management rather than some second-order interactions between speculative load execution and the SMT microarchitecture.

2. Background and Motivation

In this section, we first briefly review some commonly proposed SMT fetch policies and explain some of the program behaviors that can lead to performance degradations. We then discuss speculative execution of load instructions in the presence of earlier unresolved store addresses (speculative memory disambiguation) and its interaction with the SMT microarchitecture.

2.1. SMT Fetch Policies and Resource Stalls

The instruction fetch unit of an SMT processor has a huge influence on the overall throughput of the processor as well as the fairness of resource sharing among the different threads. On each cycle in which the processor front-end is not stalled, the fetch unit chooses one of the n threads to fetch instructions from. The decision process or algorithm for choosing which thread to fetch from is called the fetch policy. Common policies include Round-Robin and ICOUNT [26]. There are also some other policies which are described below.

While Round-Robin provides each thread with equal time on the fetch unit, it can still lead to significant reduc-

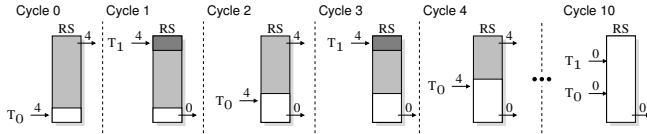


Figure 1: Shared reservation station entries in an SMT processor can be completely occupied by a stalled thread when using simple fetch policies.

tions in overall throughput. Figure 1 illustrates a simplified view of an SMT processor’s reservation station entries shared between two threads. The processor fetches from thread T_0 on even cycles and from T_1 on odd cycles. Assume that thread T_0 suffers a long-latency stall (such as a last-level cache miss) and does not have any instructions that are independent of the stalled instruction; thread T_0 will not be able to execute (0) any more instructions until the stall has been resolved. T_1 executes all four of its instructions (4) on alternate cycles. In the meantime, T_0 continues to occupy execution resources (represented by the white reservation station entries in Figure 1), and, even worse, the round-robin fetch policy continues to stuff more of T_0 ’s instructions into the out-of-order core. Eventually, the entire set of reservation stations may be occupied by T_0 ’s instructions thus stalling the entire pipeline until T_0 ’s original stall has resolved.

In one of the early SMT studies, Tullsen et al. studied a variety of SMT fetch policies to exploit the fetch unit’s choice of threads [26]. Tullsen et al. considered altering thread fetch priorities based on the per-thread counts of unresolved branches (BRCOUNT), data cache misses (MISSCOUNT), and overall instruction counts (ICOUNT). The ICOUNT policy was shown to provide the best overall throughput out of the policies evaluated, and has become a de facto standard in academic SMT studies. The fetch unit tracks the number of instructions currently in the machine for each thread and fetches from the thread with the lowest number of instructions. As stated in the original study, ICOUNT provides three major benefits: it prevents any single thread from filling up the reservation stations, provides additional priority for threads that move instructions through the processor quickly, and generally provides a more balanced ILP mix between the threads to better utilize the processor resources. The ICOUNT policy works well when the processor does not have too many low-ILP/frequently-stalling threads. Since ICOUNT does not have any explicit knowledge of stalls, any completely stalled thread will eventually consume $\frac{1}{n^{\text{th}}}$ of the reservation stations. If a given application mix contains more than a handful of such threads, then a substantial number of processor resources will be tied up, thus reducing the overall throughput of the SMT processor.

One approach to addressing this attribute of the ICOUNT policy is to anticipate the stalls. Luo et al. studied the use of branch confidence prediction [10] to control the amount of control speculation per thread [14]. The intuition is that after fetching past a few low-confidence branches, the probability of actually fetching any useful instructions rapidly tends toward zero. El-Moursy et al. also considered the prediction of data cache misses as an indicator for which threads will likely experience more stalls [8]. Neither branch confidence prediction nor load miss prediction is easy, which leaves a fetch policy based on either of these predictors vulnerable to the mispredictions. The load-miss prediction policy augments the count of predicted load misses with a count of actual load misses to help offset the difficulties in load-miss prediction. Unfortunately, when the processor discovers that a load has missed in the cache, the fetch unit may have already fetched many instructions dependent on this load miss. A hybrid load-miss predictor was also proposed in [27]. This predictor uses a majority vote to choose between predictions made by a basic predictor similar to El-Moursy’s, a gshare predictor and a gskewed predictor.

There have been several other similar cache-miss related SMT fetch policies such as STALL and FLUSH [25], DC-PRED [13] and DCache-Warn [4]. Another recently proposed policy makes use of the available memory-level parallelism (MLP) in a thread [9]. The key idea is a thread that overlaps multiple independent long-latency loads has a high-level of MLP, and therefore this thread should be allowed to allocate as many resources as needed in order to fully expose this available MLP. In the event of an isolated load miss, however, this thread will be stalled from fetching further instructions and the remaining processor resources will be available to the other threads.

2.2. Memory Dependences and SMT

In the preceding section, we described a few previously proposed techniques for predicting impending stalls before they happen. The prior work has focused primarily on branch and data-cache related stalls. Instructions in modern superscalar processors often experience a different kind of stall due to memory dependences. A load may have its address computed and be ready to issue to the data cache unit, but if there exists earlier store instructions whose addresses have not been computed, the processor will not know whether it is safe to allow the load to access the cache.

Some processors such as the Alpha 21264 [11] and the Intel Core 2 [6] allow load instructions to speculatively execute before all previous store addresses have been computed. A memory dependence predictor decides whether a load can execute as soon as its address is known, or whether it should wait until certain or all previous store addresses have been computed [5, 16, 23]. Accurate predictions allow

loads to execute earlier, but mispredictions result in costly pipeline flushes. Prior work on memory dependence prediction has demonstrated that the relationships between load and stores are highly predictable, and the predictability of these patterns has been exploited in other memory scheduling research [3, 17, 18, 22].

To the best of our knowledge, no prior work has explored the interactions between speculative memory disambiguation and simultaneous multithreading. In this section, we briefly present the results of using memory dependence prediction in an SMT processor. The full details of our simulation methodology and workloads can be found in Section 4, but in short we model a four-threaded SMT processor with and without support for speculative memory disambiguation using a variety of multi-programmed workloads. Figure 2 shows the speedup of an eight-issue SMT processor using Round-Robin (RR) and ICOUNT fetch policies over an SMT processor with no speculative memory disambiguation. The workloads include different mixes of benchmarks with low, medium or high ILP levels (L/M/H) and programs that are sensitive (S) and not-sensitive (N) to speculative load reordering. For the configurations supporting memory dependence speculation, we implemented a 4K-entry load-wait table (like that used in the Alpha 21264 [11]) and an oracle dependence predictor (predicts exactly which store a load is waiting for and the load executes immediately after the store executes) for comparison. We also ran our simulator in single-threaded mode to measure the impact of memory dependence prediction on single threaded applications in our simulator. We found that for single threaded applications, perfect memory dependence prediction provides 7% performance improvement over no memory dependence prediction.

The results of our four threaded experiments are presented in Figure 2. The realistic load-wait table (LWT) provides 2.0% and 1.0% improvements in overall throughput for RR and ICOUNT, respectively. With an unachievable oracle predictor, the overall throughput increases by 4.1% and 4.0% over the baseline RR and ICOUNT. These relatively low performance benefits may at first be somewhat surprising given the value of speculative memory disambiguation in traditional single-threaded processors. The addition of such capabilities to the Intel Core 2 microarchitecture is evidence of the value of aggressive load reordering, at least for single-threaded performance [6]. The SMT results are not unreasonable, however, since the intuition is that in a single-threaded processor, a stall due to an unresolved memory dependence can easily prevent a large number of instructions from making progress. In an SMT processor, there may be enough thread-level parallelism to largely compensate for a small number of load instructions being delayed for a few cycles. While speculative memory disambiguation and memory dependence prediction may not greatly increase the

throughput of an SMT machine, it is important to ensure high single-threaded performance. In this work, however, we concentrate on exploring other opportunities to exploit the predictability of memory dependences in SMT processors.

3. Adapting SMT Fetch Policies to Exploit Memory Dependences

The results of the previous section demonstrated that speculative memory disambiguation alone does not provide much performance benefit for SMT processors. In this section we describe a new technique that takes advantage of the fact that the memory dependences are highly predictable and uses this predictability for making instruction fetch decisions.

3.1. Proactive Exclusion

The central observation for this work is that if the timings of an instructions' stalls are predictable, then this information can be directly exploited by the SMT fetch unit to better manage the shared processor resources. We say that a fetch policy uses *proactive exclusion* if it can stop fetching from a thread before the thread's stalling condition has been exposed in the out-of-order execution core. El-Moursy and Eeckhout's fetch-gating on predicted load misses is a form of proactive exclusion [8]. In this work, we instead propose to implement proactive exclusion using a much more predictable property of programs: memory dependences.

Our PE_{mdep} (proactive exclusion based on memory dependences) is actually not a fetch policy in its own right, but rather it is an SMT fetch policy filter. That is, we can potentially apply PE_{mdep} to any other standard SMT fetch policy. PE_{mdep} starts with a memory dependence predictor, which could make use of any number of previously proposed algorithms [5, 16, 23]. When the fetch unit attempts to fetch instructions from thread i , it also consults the memory dependence predictor to find out if this thread will likely stall due to ambiguous memory dependences, as shown in Figure 3(a). If the answer from the predictor is "no dependence," then the underlying fetch policy continues operating as it otherwise would without PE_{mdep} . However if the predictor answers "dependence present," then the fetch unit finishes the current fetch and then removes thread i (T_0 in Figure 3) from the list of threads from which it can fetch instructions for (b). The fetch unit will then continue fetching, applying its original fetch policy to the current list of fetch candidates. When the load which was predicted to have a dependence issues, the out-of-order core signals the front-end that this dependence has been resolved and then thread i is reinserted into the list of fetch candidates (c). At this point, a policy like ICOUNT can help thread i "catch up" with the other threads, thus providing a reasonable amount of fairness.

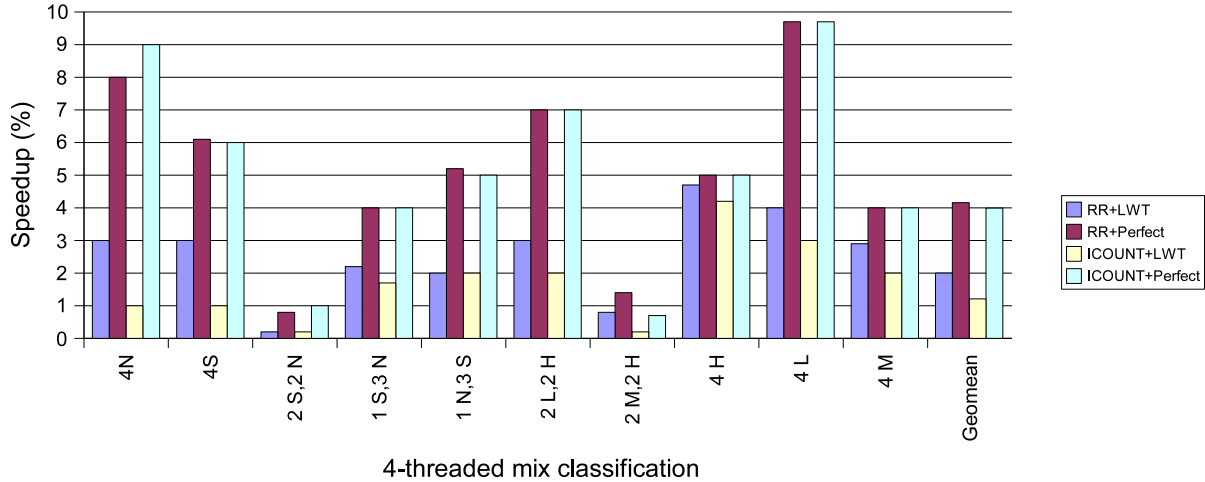


Figure 2: Impact of speculative load disambiguation on an SMT machine. The speedups (throughput improvements) are relative to SMT machines running round-robin and ICOUNT policies with no speculative memory disambiguation. LWT is the load-wait table memory dependence predictor. Perfect prediction uses an oracle predictor.

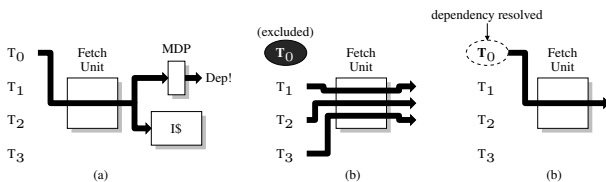


Figure 3: Demonstration of Proactive Exclusion. (a) When fetching a load, a predicted memory dependence causes the thread T_0 (b) to be excluded from the list of fetch candidates. (c) Only when the dependence has been resolved does the thread's fetching resume.

By keying fetch decisions on a highly predictable attribute, the fetch unit can easily avoid fetching the instructions that immediately follow loads with dependences. This helps to reduce the number of reservation station entries occupied by stalling operations. Following a load with a predicted memory dependence, there may be some other operations that are not data dependent on the stalled load. However, it is better for overall throughput to fetch from a different thread whose instructions are guaranteed to be independent of the stalled load rather than fetch instructions that only might be independent of the stalled load.

In the above description, we specified PE_{mdep} as checking the memory dependence predictor during fetch. For implementation purposes (and as modeled in our simulations), we do not actually consult the predictor until the instruction has been decoded and known to be a load. This could lead to a situation where a few additional instructions after the predicted-to-stall load also make it into the pipeline. There are potential negative and positive effects associated with these extra instructions. On the one hand, these instructions

may tie up additional reservation stations and reduce overall throughput. On the other hand, these extra instructions provide the thread with a little extra work ready-to-go for when the load's dependency has finally been resolved. Without these instructions, the thread may temporarily starve because the delay from dependency resolution to the fetch unit and back to the execution core can cause that thread to not execute any instructions for several cycles. Note that the number of additional instructions fetched from this thread is usually limited since the fetch unit will choose to fetch from other threads as well.

3.2. Leniency and Early Parole

At the end of the discussion of Proactive Exclusion, we discussed the potential problem of temporary thread starvation due to the delay between dependency resolution and when instructions from that thread make it back into the execution core. Figure 4(a) illustrates this scenario with a simplified view of the pipeline where a dependency has been predicted but is also quickly resolved. While a load may need to stall for ambiguous memory dependences, the time required to resolve this ambiguity may actually be fairly short. Proactively excluding the thread T_0 based on the predicted latency for instruction M hurts performance since the actual resolution latency is less than the resolution-to-fetch-to-issue delay, thereby starving T_0 for cycles $i+5$ through $i+8$ in Figure 4(a).

An interesting and useful attribute of memory dependences is that not only are their existences predictable, but their durations are predictable as well. We augment our PE_{mdep} technique with a PC-indexed delay predictor, indicated as ΔP in Figure 4(b). This structure could be im-

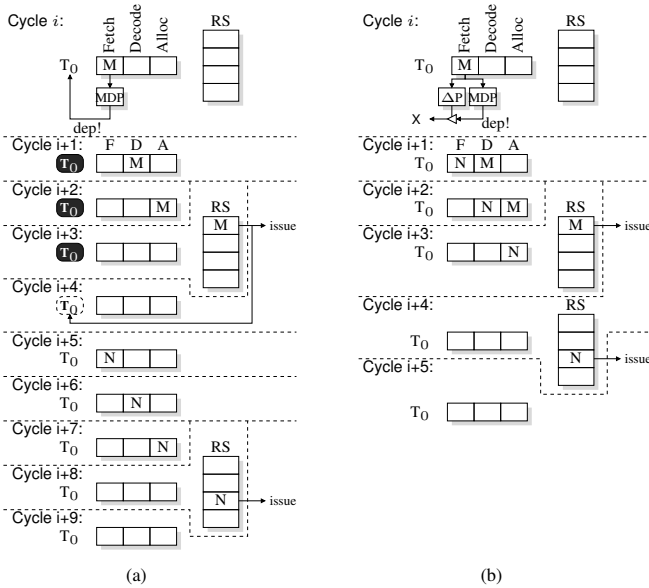


Figure 4: Timing diagram illustrating additional delays induced by (a) excluding a thread when its resolution delay is very short, and (b) avoiding its exclusion.

plemented as an additional field in the existing dependence predictor entries. If the predicted delay is less than a fixed threshold, then we forgo the proactive exclusion of this thread under the assumption that the stall will not last long enough to make exclusion worthwhile. This is the concept of *Leniency*. In this example, the load M has resolved its memory dependences by the time its dependent instruction N has made it to execute, thereby avoiding any slowdown in this thread.

We measure the resolution delay as the number of cycles between when a load instruction is ready (effective address computed) and when it actually issues. The delay predictor is updated at load commit with the new resolution delay value based on the type of delay predictor being used. Our conservative delay predictor tracks the observed delay and records the maximum delay value ever observed. To prevent the table from completely saturating with only long-latency predictions, we periodically reset the table. We also consider a more aggressive delay predictor that simply predicts whatever the last observed delay was. Finally we also model an adaptive delay predictor that predicts the delay of a thread based on the last n observed delays and then uses the average of these delays as the next predicted delay. The adaptive policy takes into account differing program phases and provides a more accurate prediction of load resolution delays. Knowing the predicted stalls can help the fetch policy to make educated decisions regarding the exclusion of a thread predicted to have a memory dependence. As explained earlier, excluding a thread predicted to have a

quickly resolving dependence can hurt performance instead of improving it. The overall goal is for the PE_{mdep} filter to be more lenient to threads with quickly resolving memory dependences.

In the case of a load with an actual long-latency resolution, PE_{mdep} (even with Leniency) still suffers from the problem of temporary thread starvation while waiting for new instructions to make their way down the pipeline. Since the dependence resolution delay is predictable, we can even avoid this temporary period of starvation for a thread. To facilitate this, we grant the excluded thread an *Early Parole* (EP) and start fetching new instructions before the actual resolution of its dependence. With accurate delay predictions, this allows the excluded thread’s instructions to arrive in the out-of-order execution core “just-in-time” as the original memory dependence has been resolved. In this fashion, we can avoid clogging the reservation stations with stalled instructions while simultaneously ensuring the timely delivery of new instructions when needed. If the fetch unit receives notification of load dependence resolution before the delay has elapsed, then it immediately returns the excluded thread to its fetch candidate list.

The implementation of Early Parole is easy to implement on top of Leniency. Given a thread’s dependence delay prediction, the fetch unit sets a counter for that thread, and decrements the counter on each subsequent cycle. When the counter reaches a threshold θ_{EP} , the fetch unit reinserts the thread into the list of fetch candidates. Note that this may occur before the thread’s memory dependence has actually been resolved. When the Leniency threshold is equal to θ_{EP} , then Leniency effectively becomes a special case of EP where the predicted delay is less than θ_{EP} and so the thread would conceptually get excluded and then immediately paroled at the same time. In our experiments we found performance was best when both techniques are simultaneously employed, and so for the remainder of this paper, the designation EP implies both Early Parole as well as Leniency. The next section presents and analyzes the impact of memory-dependence based Proactive Exclusion as well as Early Parole on SMT performance.

4. Experimental Evaluation

Figure 5 shows a summary of the primary experimental speedup over ICOUNT for PEEP. Compared to the load-miss prediction fetch policy using a simple, single-table load-miss predictor (LMP), a load-miss prediction fetch policy using a hybrid load-miss predictor (HMLP), and the MLP-based fetch policy (MLP), PEEP provides both greater overall performance/throughput (17%) as well as greater fairness (19%) as measured by the harmonic mean of weighted IPC metric [12].

In this section, after explaining our experimental methodology, we will show in steps how the individual

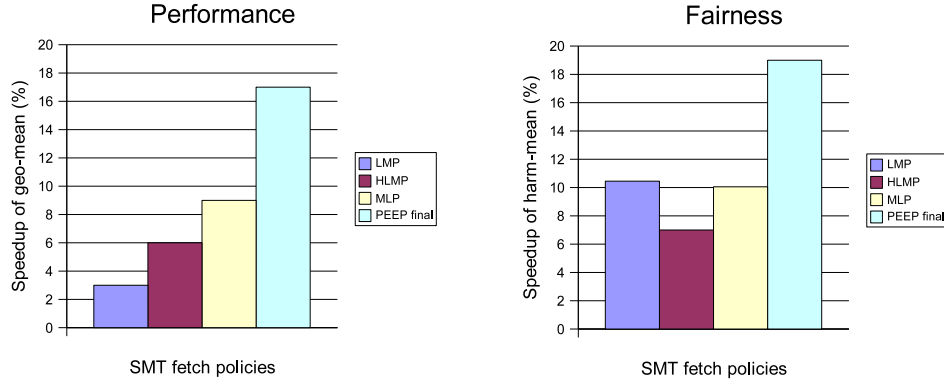


Figure 5: Performance and fairness overview of the SMT fetch policies. PEEP_{final} represents our proposed fetch policy after the previously described delay prediction optimization

Processor Width	8	Fetch-to-Issue	5 cycles
Scheduler Size (RS)	128	IL1 Cache	64KB, 2-way, 2-cycle
LSQ Size	256	DL1 Cache	32KB, 4-way, 2-cycle
ROB Size	512	L2 Cache	512KB, 8-way, 12-cycle
Integer ALU/Mult	8/4	Main Memory	200 cycles
FP ALU/Mult	8/4	ITLB	16-entry, FA
Store Ports	2	DTLB	32-entry, FA
Load Ports	4		

Table 1: Simulated four-way SMT processor configuration.

techniques of PE and EP can combine to provide this speedup.

4.1. Methodology

For all of our experiments, we used the M-Sim 2.0 simulator [19] which is based on SimpleScalar 3.0d [1]. The simulator models an SMT processor modeling up to 8 threads. Table 1 lists the parameters for our simulated SMT processor. In M-Sim’s SMT implementation, the ROB and LSQ are hard-partitioned such that with four threads running, each is constrained to only one fourth of the ROB or LSQ sizes given in the table. This is similar to the Pentium 4’s Hyperthreading implementation of SMT [15]. We modified the simulator to include speculative memory disambiguation using a memory dependence predictor based on the load-wait table (LWT) employed in the Alpha 21264 [11]. We also evaluated the final version of our design with an oracle predictor to determine the upper bound on the performance and to determine if the load-wait table predictor is too conservative.

We ran four-thread mixes from a variety of benchmark suites. Our benchmarks belong to the integer (I) and floating point (F) SPEC2000 suites, MediaBench (M), MiBench (E) and the pointer-intensive applications (P) from Wisconsin [2]. The details of the four-threaded mixes are given in Table 2. For each benchmark, we classify the application as either exhibiting high (H), medium (M) or low-ILP (L), and memory dependence-sensitive (S) or not (N). The

ILP-based workload mixes and the classification of memory dependence sensitivity are the same as those used in some other previous studies [20, 21, 23].

Per-workload performance numbers are reported as relative improvements over a baseline machine using the ICOUNT fetch policy with speculative memory disambiguation enabled. Aggregate performance numbers use the geometric mean, which has the property that the geometric mean of speedups is always equal to the speedup of the geometric means. We also quantify the impact of our proposal on overall fairness. To measure fairness, we considered the the harmonic mean of the weighted IPCs for all of the fetch policies. This metric provides a measure of both performance and fairness and is typically used in multi-threading research studies to quantify fairness [12]. We also report the standard deviation of performance degradation experienced by each thread relative to its stand alone IPC (i.e., if the thread had the entire processor to itself). If the standard deviation is large, then that means that some threads experienced larger slowdowns than others, thereby implying that the situation is unfair. This is a stricter measure of fairness as it does not explicitly account for overall system throughput.

For all configurations, we use a load-wait table dependence predictor with 4096 one-bit entries [11]. The table is reset every one million cycles to ensure that the predictor does not become too conservative. In configurations employing a delay predictor, we use a 4K-entry table of 7-bit counters, totaling to 4KB of state overhead when counting both the delay and dependence predictors.

4.2. Proactive Exclusion Only

We first present results for Proactive Exclusion (PE) by itself without the effects of Early Parole. All speedup results in Figure 6 are relative to the performance of the standard ICOUNT fetch policy. The load-miss predictor fetch policies have overall speedups of 3% and 6% for LMP and

Mix Classification	Benchmarks	Application Suite	Mix Classification	Benchmarks	Application Suite
(4 N)-1	art,bzip2,patricia,quake	F/I/E/F	(3 S,1 N)-1	vortex3,perbm3,eonc,ampp	I/I/I/F
(4 N)-2	ampp,applu,swim,wupwise	F/F/F/F	(3 S,1 N)-2	eonc,gcc,mesa2,art	I/I/F/F
(4 N)-3	mcf,wupwise,epic,dijkstra	I/F/M/E	(3 S,1 N)-3	fma3d,vortex1,vpr1,ampp	F/I/I/F
(4 N)-4	patricia,lucas,gap,fft	E/F/I/E	(3 S,1 N)-4	crafty,tiffdither,vortex1,patricia	I/E/I/E
(4 S)-1	apsi,eonc,gcc2,perlbmk1	F/I/I/I	(4 L)-1	mcf,quake,art,lucas	I/F/F/F
(4 S)-2	gcc,yacr,twolf,tiffdither	I/P/I/E	(4 L)-2	twolf,vpr2,swim,parser	I/I/F/I
(4 S)-3	gcc2,perlbmk1,eonr,vortex1	I/I/I/I	(4 M)-1	applu,ampp,mgrid,galgel	F/F/F/F
(4 S)-4	twolf,gcc2,anagram,jpegencode	I/I/P/M	(4 M)-2	gcc,bzip2,eonc,apsi	I/I/I/F
(2 S,2 N)-1	vpr1,yacr,wupwise,mcf	I/P/F/I	(4 H)-1	facerec,crafty,perlbmk1,gap	F/I/I/I
(2 S,2 N)-2	apsi,anagram,patricia,dijkstra	F/P/E/E	(4 H)-2	wupwise,gzip3,vortex1,mesa1	F/I/I/M
(2 S,2 N)-3	bc-fact,bc-primers,swim,ampp	P/P/F/F	(2 L,2 H)-1	mcf,quake,mesa2,vortex2	I/F/F/I
(2 S,2 N)-4	gs,parser,fft,galgel	E/I/E/F	(2 L,2 H)-2	parser,swim,crafty,perlbmk3	I/I/I/I
(3 N,1 S)-1	galgel,art,ampp,crafty	F/F/F/I	(2 L,2 M)-1	parser,swim,gcc3,bzip2	I/F/I/I
(3 N,1 S)-2	mcf,lucas,swim,vortex1	I/F/F/I	(2 L,2 M)-2	art,lucas,galgel,gcc	F/F/F/I
(3 N,1 S)-3	art,bzip2,lucas,eonk	F/I/F/I	(2 M,2 H)-1	gzip3,wupwise,fma3d,apsi	I/F/F/F
(3 N,1 S)-4	applu,galgel,mgrid,anagram	F/F/F/P	(2 M,2 H)-2	vortex3,mesa2,mgrid,eonr	I/F/F/I

Table 2: Multi-programmed application mixes used for the four-threaded experiments.

HMLP. Load misses are relatively difficult to predict, and so while this approach provides some performance benefit, many mispredicted load misses lead to underutilization of the processor resources. The MLP-aware policy is able to expose a higher level of parallelism, which proves to be a better technique than using load-miss prediction, yielding a speedup of 9% over ICOUNT. The general trends are consistent with previously reported results. Applying our memory dependence-based proactive exclusion, the PE_{mdep} fetch policy performs quite well when compared to ICOUNT providing a geometric mean performance benefit of nearly 13% with five mixes showing over 30% performance improvement. Only one mix posts a performance degradation: the (2L-2H)-2 mix suffers from a 4% performance decrease, although this is avoided when we include the delay predictor. PE_{mdep} provides more robust performance gains, while LMP sometimes exhibits some significant losses.

4.3. Proactive Exclusion with Early Parole (PEEP)

Some of the benefit of reducing resource contention through PE is offset by the additional delay required to get new instructions from a stalled thread back into the pipeline. The use of Early Parole (and Leniency) significantly helps the problem of a slow restart after dependence resolution. Figure 7 shows the results of adding EP on top of PE_{mdep} . We evaluate delay predictors with conservative (largest delay observed), aggressive (most recent delay observed) and adaptive behaviors. Leniency is invoked if the predicted delay is less than or equal to five cycles (the length of the front-end pipeline), and Early Parole is granted five cycles before the predicted delay has elapsed. The conservative delay predictor provides an overall benefit of 16% over ICOUNT.

Our use of delay prediction only affects the fetch unit, and therefore inaccurate delay predictions only affect resource contention. As a result, the conservative delay predictor sometimes induces more stalling than is necessary. Using the aggressive predictor provides a slight performance advantage with effectively no additional cost or complexity as compared to the conservative approach. PEEP us-

ing an adaptive delay predictor provides an overall performance gain of 17%. This slight performance improvement, however, is likely not worth the additional hardware cost of tracking multiple delays per load. It is also interesting to note that the one mix that showed a 4% performance degradation without the use of Early Parole now shows a 16% performance improvement with PEEP.

The performance benefits vary among the different workload mixes. For example, when all applications are not sensitive to memory dependencies, such as (4N)-2, then there are practically no long-latency load dependence resolutions for PEEP to take advantage of. In this situation, while we do not gain much performance, PEEP gracefully degrades to the underlying ICOUNT policy. The mix (4L)-1 achieves very large speedups (70%). On further analysis, we observed that there is a huge amount of load port pressure when the ICOUNT fetch policy is used. PEEP and MLP both reduce memory accesses from threads which are predicted to stall, thereby decreasing port contention. The port contention effect is further amplified by M-Sim’s accurate modeling of latency-misprediction-induced scheduler replays. Any load directly dependent on an earlier mis-scheduled load (as occurs with pointer-chasing behaviors), may replay one or more times, with each attempt consuming more load issue slots. When the number of load ports was increased from four to eight for the ICOUNT policy, this performance difference was reduced.

There are some application mixes which need some further detailed analysis. The first of these peculiar mixes is (2S,2N)-1. All configurations involving PEEP manage to provide a performance benefit in excess of 30%. The memory dependence sensitive benchmarks in this mix ‘yacr’ and ‘vpr’ have very predictable dependences and even the simple dependence predictor is able to exploit these. Additionally one of the applications in this mix ‘mcf’ exhibits considerable levels of MLP which helps the MLP-aware algorithm to provide good performance, too. In the case of (3N,1S)-2, we have a good example of the potential performance benefit by accounting for memory dependence pre-

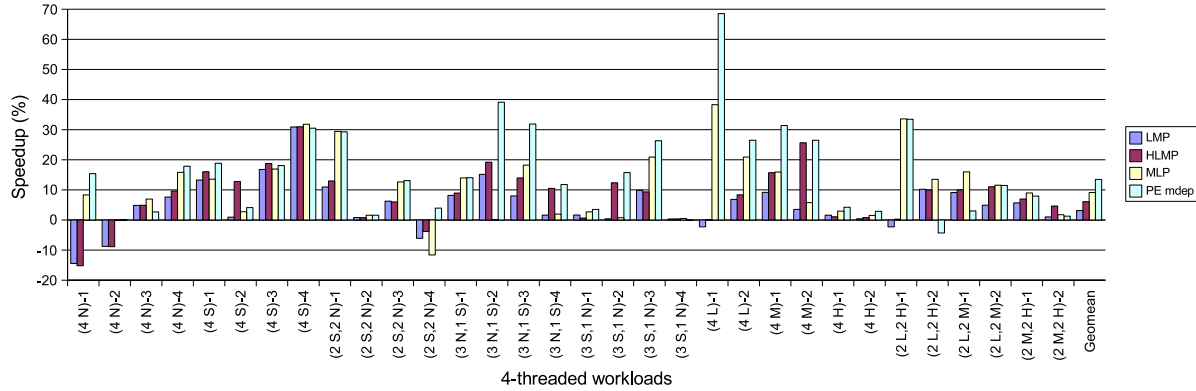


Figure 6: Speedup over ICOUNT of the basic SMT fetch policies as well as the Proactive Exclusion policy based on Memory Dependences without any delay prediction.

diction to guide fetch of threads. ‘Vortex’ happens to be a benchmark that has a very high memory dependence sensitivity whereas the other applications like ‘mcf’ and ‘lucas’ are highly insensitive to memory dependences. What this means is that the PEEP algorithm is able to very accurately fetch from these insensitive threads while ‘vortex’ is having a stall and then switch the fetching when the stall resolves. The delay predictors are able to further improve this performance since even the memory dependence resolution delays in vortex are predictable. We observed very few fetch stalls in the processor when the PEEP algorithm was being employed. The MLP-aware algorithm though is unable to exploit the maximum potential of this phenomenon.

To test the upper limit on the performance that PEEP could offer we evaluated both the ICOUNT fetch policy as well as PEEP with an oracle memory dependence predictor. We found that PEEP provides a 19% performance improvement over the ICOUNT fetch policy when an “unrealistic” oracle predictor is used which is just 2-3% (depending on the delay predictor) more than when a load-wait table based predictor is used. This indicates that while the load-wait table is a slightly conservative predictor, it is probably sufficient in an SMT processor. These oracle predictor results are also important because they show that the main benefit of PEEP comes from being able to accurately predict the memory disambiguation stalls and their related latencies, rather than the fetch policy somehow indirectly reducing the number of ordering violations. Since the performance of the load-wait table based predictor and the oracle predictor is so close, this shows that the LWT predictor does not cause any significant artificial stalls by making loads wait unnecessarily on independent stores. If this was the case the performance using the oracle predictor (which as described earlier only identifies true dependences) would have been higher.

An interesting observation is that while memory disam-

biguation delays are not as long as load miss delays on average, some resolution delays we noted were considerably long. The reason for this is that some store addresses have a dependence on earlier loads that miss in the cache, which in turn can force a later load to stall until the original miss returns and the store can compute its address.

One of the reasons why PEEP performs better than either the LMP or MLP approaches is that proactive exclusion does not kick in until we reach the load with the long predicted dependence resolution latency. Consider a load A that must access main memory, followed by a dependent store B, and then another load C. Store B cannot compute its address until A has returned from memory. In the meantime, if Load C has been predicted to *not* have a dependency on Store B, then it (and its dependents) can execute and make forward progress. In the case of LMP, the fetch policy would have stopped fetching from this thread immediately after fetching Load A, thereby missing out on the ILP available further in the instruction stream. If Load A is an isolated miss, then the same scenario would apply to the MLP-based policy as well. In the case that Load C has been predicted to be dependent on Store B, then the fetch policy would stop at Load C. The pipeline would, however, already contain some instructions past the original stalled Load A. This indirectly creates an effect somewhat similar to our Early Parole in that a small amount of additional work (only up to Load C) is already available in the out-of-order core to prevent starvation when Load A returns from memory. As execution finishes up on these instructions, the Early Parole effect for Load C should allow more instructions to arrive in the execution core just in time to prevent starvation when Load C executes.

4.4. Fairness Results

When dealing with multi-programmed workloads, high performance and throughput are not always the only impor-

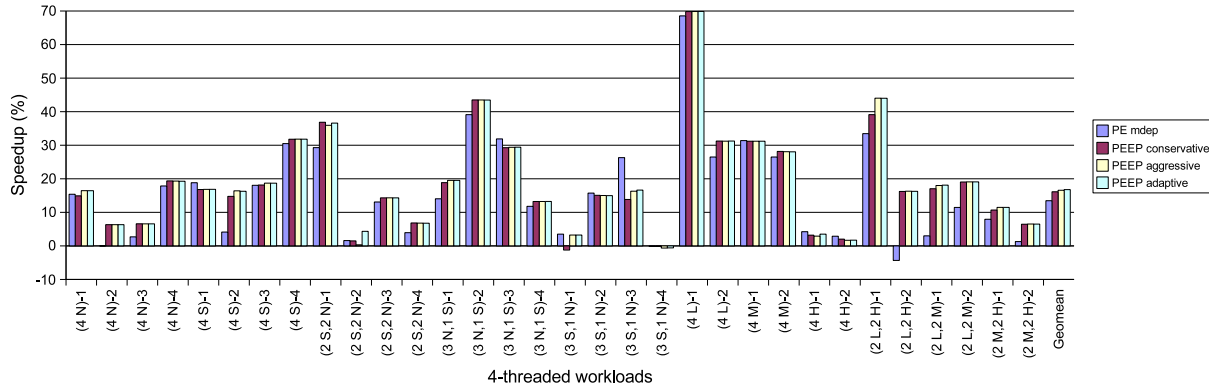


Figure 7: Combining Proactive Exclusion and Early Parole with different delay predictors.

tant metrics. In many systems, maintaining fairness among threads is also of great importance. A greedy scheme that only benefits high-ILP applications or applications having no memory dependences may over-penalize and starve other threads. This is one of the reasons why ICOUNT is considered to be a good policy for an SMT processor. ICOUNT provides relatively high throughput while ensuring that even slow-moving threads get their fair share of resources. Since PEEP runs on top of ICOUNT (or potentially any other fetch policy), PEEP effectively inherits ICOUNT's fairness properties. Immediately after an excluded thread has been reinserted into the candidate list, it would likely have fewer instructions in the pipeline, and therefore the underlying ICOUNT policy will heavily favor fetching from this thread. Figure 8 displays the harmonic mean of weighted IPCs (HMWIPC) for the four-threaded workloads. PEEP delivers greater performance than the other policies and results in better fairness properties as measured by the HMWIPC (20% higher than ICOUNT). The standard deviation of speedup (SDS) metric is a stricter measure of fairness because it does not take performance into account like HMWIPC. Our results indicate that both ICOUNT and PEEP running on top of ICOUNT exhibit very low slowdown standard deviations (0.11 and 0.17 for ICOUNT and PEEP, respectively).

5. PEEP on an SMT Processor without Speculative Memory Disambiguation

The PEEP results presented so far demonstrate strong performance gains for a relatively simple modification to basic SMT fetch policies. However, the idea is based on the assumption that an SMT processor has support for speculative memory disambiguation in the first place. While memory disambiguation and load-store reordering are known to improve performance and would likely be included in future processors, there have also been SMT based processors in the past which have not supported it. For example, the Intel

Pentium 4 with Hyperthreading supports a version of SMT, but does not execute loads out-of-order ahead of earlier unresolved store addresses [6].

In this section, we evaluate an alternative design based on the main PEEP idea. We allow the SMT processor to make dependency predictions and consequently predict load dependence resolution delays, and we use PEEP to control the fetch policy. However, *controlling fetch is the only thing these predictions are used for*. The out-of-order execution core does *not* allow loads to issue in the presence of earlier unresolved stores, even if the load has a predicted delay of zero. Delay still has the same definition as in Section 3; that is, the time from when a load's address has been computed until it actually issues. Any predicted non-zero delay counts as a predicted memory dependence, but Leniency is still used to prevent exclusion when the delay is less than the given threshold (five cycles for our experiments). In an SMT processor without speculative memory disambiguation, load instructions will on average have to wait longer for all earlier store addresses to resolve. Employing PEEP in this context does not attempt to reduce this latency; it cannot. Rather, PEEP only targets the reduction of scheduler congestion effects when such stalls are present. The intuition here is that a large fraction of the performance benefit comes from being able to predict when these stalls occur and more importantly how long they take to resolve. These two pieces of information are what guide the processor to make intelligent decisions about which threads to fetch from, which threads to throttle and for how long. We refer to this version of PEEP without speculative load reordering as PEEP*.

Figure 9 shows the performance of PEEP* compared to a baseline SMT processor using ICOUNT, also without speculative memory reordering. While not allowing loads to issue out-of-order with respect to earlier unresolved stores potentially reduces ILP, it helps to avoid pipeline flushes on mispredictions. The relative results are very close to

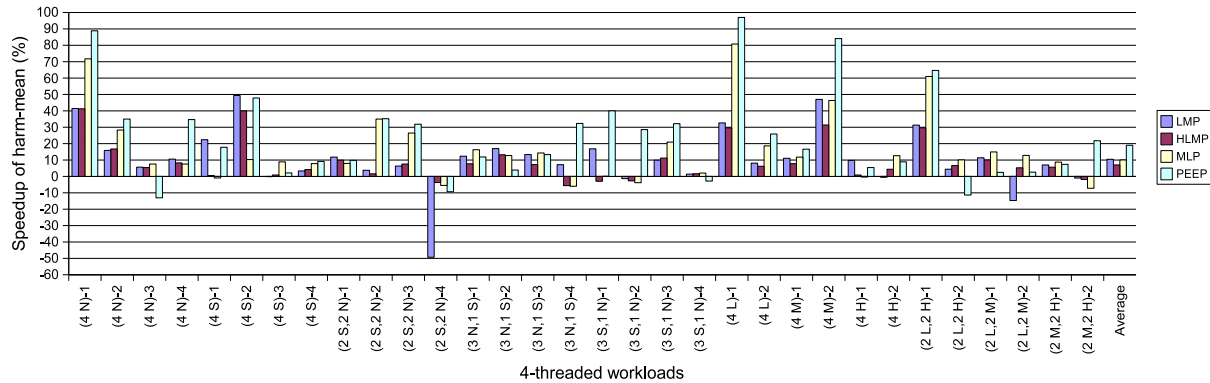


Figure 8: Fairness as measured by the harmonic mean of weighted IPC.

the original PEEP design indicating that a fetch policy that is cognizant of impending memory disambiguation stalls helps to improve the throughput of an SMT machine considerably. From these results, we draw two conclusions. First, predicted memory dependences are very effective indicators of impending stalls that an SMT fetch policy should be aware of. Second, the benefits of our proposed technique are purely a result of better fetch management and they are not dependent on any explicit support for speculative memory disambiguation in the out-of-order core.

6. Sensitivity Analysis

Our baseline four-threaded SMT microarchitecture was chosen to provide a moderate level of aggressiveness. We now study the impact of scaling down the processor configuration and workloads, as well as further scaling up the microarchitecture. We first consider a smaller processor operating only on two-threaded workloads. Table 3 shows these workloads, which combine the same classes of applications as before. The processor configuration is similar to that described earlier in Table 1, with per-thread resources (e.g., ROB entries) reduced to handle only two threads. Figure 10 shows the corresponding performance results for LMP, HLMP, MLP, and PEEP/PEEP*.

By its nature, PEEP throttles the fetch of a thread predicted to have dependencies while allowing other independent threads to utilize all the processor resources. This means that with more threads, PEEP has more opportunities to fetch from a non-stalled thread. Conversely, with fewer threads such as in the case of our two-thread workloads, if one (or worse both!) threads suffer from a memory-dependence related stall, then there are far fewer options for finding useful work to cover the dependency resolution latency. As a result we do not expect PEEP to provide the same level of performance gains in this scenario. Figure 10 shows, however, that the results are not too discouraging either. The reason is that not all stalls occur simultaneously. If only one of the two threads has a predicted memory de-

pendence, then PEEP allows the other thread to make use of practically all of the processor’s resources. Only when both threads have predicted memory dependences will the processor’s throughput be severely reduced, which does not happen too often.

We also evaluated PEEP/PEEP* on a larger, more aggressive SMT microarchitecture. We modeled a configuration containing a 2048-entry ROB, 512-entry LSQ, 300-entry RS and 10 memory ports. Larger instruction windows allow the processor to consider more instructions per thread, which in turn increases the likelihood that some load instructions will be stalled waiting on ambiguous memory dependences. In this context, it is that much more critical for the SMT fetch policy to account for these dependences to avoid tying up processor resources with those instructions in the forward slices of the stalled loads. Using PEEP and PEEP* for this large SMT processor configuration yield average throughput improvements of 20% for both approaches. The per-workload results are omitted due to space constraints, but the overall trends are very similar to those reported in Figures 7 and 9. As machine sizes scale upward, however, the difference between PEEP and MLP decreases as larger windows expose more memory-level parallelism.

7. Summary

In this work, we have studied the concepts of Proactive Exclusion (PE) to remove threads from an SMT processor’s fetch unit’s work list when threads are likely to experience long-latency stalls. We also proposed an Early Parole (EP) mechanism to restart fetching of previously excluded threads in an anticipatory fashion such that the instructions arrive at the out-of-order execution core right as the previous stall resolves. In particular, we took advantage of the predictability of load-store memory dependences as well as the predictability of their resolution delays. While this work focused on memory dependence prediction, the PE and EP techniques can potentially be applied to any stalls in SMT

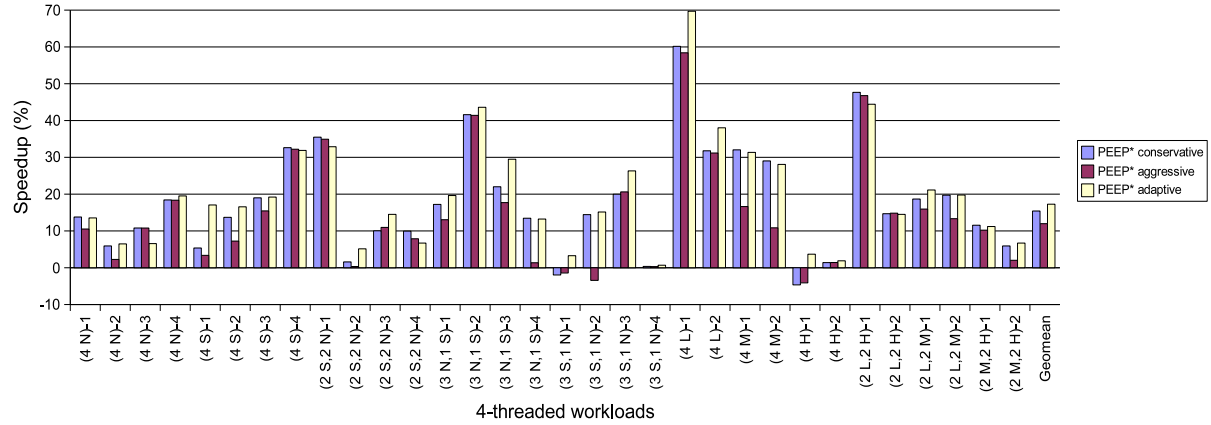


Figure 9: Benefit of memory dependence prediction without speculative load disambiguation.

Mix Classification	Benchmarks	Application Suite	Mix Classification	Benchmarks	Application Suite
(2 N)-1	art,bzip2	F/I	(2 L)-1	mcf,quake	I/F
(2 N)-2	ammp,applu	F/F	(2 L)-2	twolf,vpr2	I/I
(2 N)-3	mcf,wupwise	I/F	(2 M)-1	applu,ammp	F/F
(2 N)-4	patricia,lucas	E/F	(2 M)-2	gcc,bzip2	I/I
(2 S)-1	gcc2,perlbmk1	I/I	(2 H)-1	facerec,crafty	F/I
(2 S)-2	gcc,twof	I/I	(2 H)-2	wupwise,gzip3	F/I
(2 S)-3	apsi,eonc	F/I	(1 L,1 H)-1	mcf,mesa2	I/F
(2 S)-4	gcc,yacr	I/P	(1 L,1 H)-2	parser,crafty	I/I
(1 S,1 N)-1	vpr1,wupwise	I/F	(1 M,1 H)-1	gzip3,fma3d	I/F
(1 S,1 N)-2	apsi,patricia	F/E	(1 M,1 H)-2	vortex3,mgrid	I/F
(1 S,1 N)-3	bc-fact,swim	P/F	(1 L,1 M)-1	parser,gcc3	I/I
(1 S,1 N)-4	parser,galgel	I/F	(1 L,1 M)-2	art,galgel	F/F

Table 3: Two-threaded workloads.

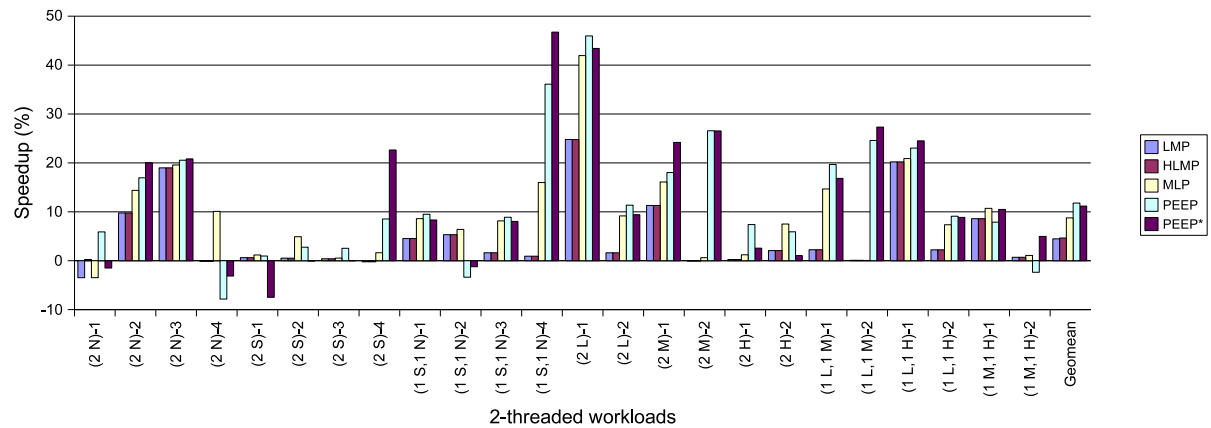


Figure 10: Performance of the SMT fetch policies for two-threaded workloads.

processors that are easily predictable. In this work we analyze the performance of PEEP with both a simple load-wait table based predictor as well as an oracle or a perfect predictor. Analyzing the gamut of predictors helps in determining efficient performance/cost trade offs. An important and surprising result of this work is that simply identifying the existence and duration of the dependence-induced stalls also provides significant improvements in overall SMT bandwidth. This is an interesting result since the original idea of memory dependence prediction was developed only to allow for load-store reordering, and we now argue that given a memory dependence predictor, one can extend its use to guide efficient instruction fetch in an SMT processor.

Acknowledgments

This research was sponsored in part by equipment and funding donations from Intel Corporation. We are grateful for the constructive feedback provided by the anonymous reviewers.

References

- [1] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Micro Magazine*, pages 59–67, February 2002.
- [2] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–301, Orlando, FL, USA, June 1994.
- [3] Fernando Castro, Luis Pinuel, Daniel Chaver, Manuel Prieto, Michael Huang, and Francisco Tirado. DMDC: Delayed Memory Dependence Checking through Age-Based Filtering. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 297–306, Orlando, FL, December 2006.
- [4] Francisco J. Cazorla, Alex Ramierz, Mateo Valero, and Enrique Fernández. DCache Warn: an I-Fetch Policy to Increase SMT Efficiency. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, pages 74–83, Santa Fe, NM, USA, April 2004.
- [5] George Z. Chrysos and Joel S. Emer. Memory Dependence Prediction Using Store Sets. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 142–153, Barcelona, Spain, June 1998.
- [6] Jack Doweck. Inside Intel Core Microarchitecture and Smart Memory Access. White paper, Intel Corporation, 2006. <http://download.intel.com/technology/architecture/sma.pdf>.
- [7] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous Multithreading: A Platform for Next-Generation Processors. *IEEE Micro Magazine*, 25(5):12–19, September–October 1997.
- [8] Ali El-Moursy and David Albonesi. Front-End Policies for Improved Issue Efficiency in SMT Processors. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture*, pages 185–196, Anaheim, CA, USA, February 2003.
- [9] Stijn Eyerma and Lieven Eeckhout. A Memory-Level Parallelism Aware Fetch Policy for SMT Processors. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, pages 27–36, Phoenix, AZ, USA, February 2007.
- [10] Erik Jacobson, Eric Rotenberg, and James E. Smith. Assigning Confidence to Conditional Branch Predictions. In *Proceedings of the 29th International Symposium on Microarchitecture*, pages 142–152, Paris, France, December 1996.
- [11] R. E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro Magazine*, 19(2):24–36, March–April 1999.
- [12] Manoj Franklin Kun Luo, Jayanth Gummaraju. Balancing throughput and fairness in SMT processors. In *Proceedings of the 15th International Symposium on Performance Analysis of Systems and Software*, pages 9–16, Tucson, AZ, USA, November 2001.
- [13] C. Limousin, J. Sebot, A. Vartanian, and N. Drach-Temam. Improving 3D Geometry Transformations on a Simultaneous Multithreaded SIMD Processor. In *Proceedings of the International Conference on Supercomputing*, pages 236–245, Sorrento, Italy, June 2001.
- [14] Kun Luo, Manoj Franklin, Shubuhendu Mukherjee, and Andre Seznec. Boosting SMT Performance by Speculation Control. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, pages 9–16, San Francisco, CA, USA, April 2001.
- [15] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton. Hyper-Threading Technology and Microarchitecture. *Intel Technology Journal*, 6(1), February 2002.
- [16] Andreas Moshovos. *Memory Dependence Prediction*. PhD thesis, University of Wisconsin, 1998.
- [17] Tingting Sha, Milo M. K. Martin, and Amir Roth. Scalable Store-Load Forwarding via Store Queue Index Prediction. In *Proceedings of the 38th International Symposium on Microarchitecture*, pages 159–170, Barcelona, Spain, November 2005.
- [18] Tingting Sha, Milo M. K. Martin, and Amir Roth. NoSQ: Store-Load Forwarding without a Store Queue. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 285–296, Orlando, FL, December 2006.
- [19] Joseph Sharkey. M-Sim: A Flexible, Multithreaded Architectural Simulation Environment. CS-TR 05-DP01, State University of New York at Binghamton, Department of Computer Science, 2005.
- [20] Joseph Sharkey, Deniz Balkan, and Dmitry Ponomarev. Adaptive Reorder Buffers for SMT Processors. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, pages 244–253, Seattle, WA, USA, September 2006.
- [21] Joseph Sharkey and Dmitry Ponomarev. Efficient Instruction Schedulers for SMT Processors. In *Proceedings of the 12th International Symposium on High Performance Computer Architecture*, pages 293–303, Austin, TX, USA, February 2006.
- [22] Samantika Subramaniam and Gabriel H. Loh. Fire-and-Forget: Load/Store Scheduling with No Store Queue At All. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 273–284, Orlando, FL, December 2006.
- [23] Samantika Subramaniam and Gabriel H. Loh. Store Vectors for Scalable Memory Dependence Prediction and Scheduling. In *Proceedings of the 12th International Symposium on High Performance Computer Architecture*, pages 64–75, Austin, TX, USA, February 2006.
- [24] Dean Tullsen, Susan Eggers, and Henry Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 392–403, Santa Margherita Ligure, Italy, June 1995.
- [25] Dean M. Tullsen and J. Brown. Handling Long-Latency Loads in a Simultaneous Multithreaded Processor. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 318–327, Austin, TX, USA, December 2001.
- [26] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 191–202, Philadelphia, PA, USA, May 1996.
- [27] Adi Yoaz, Mattan Erez, Ronny Ronen, and Stephan Jourdan. Speculation Techniques for Improving Load Related Instruction Scheduling. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 42–53, Atlanta, GA, USA, June 1999.