Load Latency Tolerance In Dynamically Scheduled Processors

Srikanth T. Srinivasan and Alvin R. Lebeck
Department of Computer Science
Duke University
Durham, North Carolina 27708 USA
{sri,alvy}@cs.duke.edu

Abstract

This paper provides quantitative measurements of load latency tolerance in a dynamically scheduled processor. To determine the latency tolerance of each memory load operation, our simulations use flexible load completion policies instead of a fixed memory hierarchy that dictates the latency. Although our policies delay load completion as long as possible, they produce performance (instructions committed per cycle (IPC)) comparable to an ideal memory system where all loads complete in one cycle. Our measurements reveal that to produce IPC values within 8% of the ideal memory system, between 1% and 62% of loads need to be satisfied within a single cycle and that up to 84% can be satisfied in as many as 32 cycles, depending on the benchmark and processor configuration. Load latency tolerance is largely determined by whether an unpredictable branch is in the load's data dependence graph and the depth of the dependence graph. Our results also show that up to 36% of all loads miss in the level one cache yet have latency demands lower than second level cache access times. We also show that up to 37% of loads hit in the level one cache even though they possess enough latency tolerance to be satisfied by lower levels of the memory hierarchy.

1 Introduction

Many of today's microprocessors use dynamic scheduling [17,18] to maximize the number of instructions issued per cycle. By buffering instructions that are waiting for their operands and executing other independent instructions out of order, the processor is able to tolerate some long latency operations—including cache misses. To find enough independent instructions, most processors employ sophisticated branch prediction mechanisms [13, 21] and allow speculative execution [4, 9, 19], committing results only when the true outcome of a branch is known.

Unfortunately, because of finite resources, data dependencies and imperfect branch prediction, some operations must complete quickly to maximize processor performance. Consider a processor capable of issuing 8 instructions per cycle and a 10 cycle level two cache access latency. In the time it takes the level two cache to satisfy a load request, the processor could issue up to 80 instructions. If many of these instructions are dependent on the value returned by the load and there is insufficient buffer space because of previous long latency operations, the 10 cycle load operation would cause the processor to stall. In contrast, if many of the instructions are independent and there is sufficient buffer space, their execution could overlap with the load, and not stall the processor. Load latency tolerance exists in the latter case, when a memory reference can take many cycles to complete without adversely affecting performance.

The first contribution of this paper is to present a quantitative evaluation of load latency tolerance in a dynamically scheduled processor. Using SimpleScalar [2] we measure individual load instruction latency tolerance by forcing their completion such that the number of instructions committed per cycle (IPC) is comparable to an ideal memory system that satisfies all requests in a single cycle. We evaluate a variety of polices to force load completion in an effort to balance high IPC values with long load latencies. We find that using mispredicted branches and the depth of a load's dependence graph to determine when loads should complete, produces IPC values within 8% of the ideal memory system, while yielding noticeable latency tolerance.

Our measurements, on an 8 issue processor that can have up to 256 instructions in flight, show that between 13% and 62% of the loads in our benchmarks need to complete in one cycle and that 58% to 98% must complete in 8 cycles. Reducing the issue width to 4, reduces the number of one cycle loads to between 1% and 46% and the number of 8 cycle loads to 5% to 88%. These results show that many loads could be satisfied with latencies comparable to second-level cache hit times.

This work supported in part by NSF CAREER Award MIP-97-02547, DARPA Grant DABT63-98-1-0001, NSF Grants CDA-97-2637 and CDA-95-12356, Duke University, and an equipment donation through Intel Corporation's Technology for Education 2000 Program. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
2005		**-**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Load Latency Tolerance In Dynamically Scheduled Processors	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Defense Advanced Research projects Agency,3701 North Fairfax Drive,Arlington,VA,22203-1714**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**12**	

The second contribution of this paper is an evaluation of the match between the latencies incurred in a traditional memory hierarchy and an application's inherent latency demands. We find that between 2% and 36% of loads requiring latency below 8 cycles miss in the first level cache, depending on the cache size and application. Furthermore, our results reveal that between 2% and 37% of loads that hit in the first level cache have enough latency tolerance that they could be satisfied by lower levels of the memory hierarchy.

The remainder of this paper is organized as follows. Section 2 provides background information. Section 3 describes our technique for measuring the available load latency tolerance. Our experimental methodology is presented in Section 4, and Section 5 presents our results. Section 6 concludes this paper.

2 Background

Superscalar processors maximize serial program performance by issuing multiple instructions per cycle. Each cycle the processor attempts to issue up to *issue-width* instructions. One of the most important aspects of these systems is identifying independent instructions that can execute in parallel.

2.1 Scheduling, Prediction, and Speculation

In order to identify and exploit instruction level parallelism, most of today's processors employ dynamic scheduling, branch prediction, and speculative execution. Dynamic scheduling is an all hardware technique for identifying and issuing multiple independent instructions in a single cycle. The hardware looks ahead by fetching instructions into a buffer—called an *issue window*—from which it selects instructions to issue to the functional units. Instructions are issued only when all their operands are available, and independent instructions can execute out-of-order. Results of instructions executed out-of-order are committed to the register file in program order.

The issue window is filled with instructions from several basic blocks by predicting the direction of conditional branches [3, 6, 10, 12, 14]. Furthermore, the instructions from the predicted path are speculatively executed, hoping the branch prediction is correct. These instructions can not update the architectural state of the processor until the true outcome of the branch is determined. While waiting for the branch computation, these instructions occupy valuable buffer space potentially reducing the issue rate.

The above techniques are very effective for well-behaved programs with short-latency operations. However, long latency operations, such as load cache misses, can reduce their effectiveness for the following reasons:

1. **Data Dependencies**: If the operands of instruction *i* are produced by a previous instruction *j*, then *i* is dependent on *j*, and *i* can begin execution only after *j* has completed. Clearly, instruction *i* can not be issued in the same cycle as *j*. While looking ahead, if most of the newly dispatched instructions are dependent on earlier issued instructions waiting to complete, even a dynamically scheduled processor will be unable to identify ready instructions to execute, and the processor utilization will go down.
2. **Finite Resources**: The number of instructions the processor can look ahead to identify independent instructions is limited by the number of available entries in the issue window. Dependent instructions waiting for the result of a load and independent instructions waiting to commit their results occupy entries. For long latency operations, the window could become full and stall the processor.
3. **Branch Misprediction**: When the predicted outcome of a branch is incorrect, the work performed by the processor while executing along the incorrect path is useless.[1] If computation of the true branch outcome is delayed because of a cache miss, the processor could waste many cycles before the misprediction is detected.

The remainder of this paper examines load latency tolerance in the context of the above potential limitations.

3 Measuring Load Latency Tolerance

This section presents the policies we use to determine the latency tolerance of load instructions. Our primary goal is to quantify the amount of latency tolerance in dynamically scheduled processors. We also want to determine if there is variation among individual load latency tolerance. That is, do some loads require fast servicing while others can be satisfied in longer amounts of time without degrading performance? Finally, we want to evaluate the match between a load's latency tolerance and the latency it incurs in a conventional cache memory hierarchy.

We compute the latency tolerance of a load by measuring the number of cycles that elapse between the time the load is issued and the time it completes. A load is issued to the memory system when its effective address is available, and it completes when the referenced data is available for use by dependent instructions. In a traditional memory system, a load's latency depends primarily on which level in the memory hierarchy satisfies the request.

1. Recent techniques show how to reuse some of the results obtained during speculative execution [15].

Our goal is to determine how long a load can be outstanding without causing degradation in performance. Hence, we do not complete a load as long as the processor is able to do useful work by looking ahead and executing independent instructions. In particular, we want our measured latencies to reflect a program execution that achieves IPC close to that of an ideal memory system, where all references complete in one cycle.

Previous studies, that examined latency tolerance in decoupled architectures [8], analyzed the effects of increasing memory latency for systems both with and without caches. In the systems without caches, this produces a uniform increase in latency for all memory accesses. This type of analysis can provide some insight into latency tolerance. However, it is an all or nothing approach where every load has the same cost, and is suitable only for specific memory system designs. Similarly, increasing the memory latency in cache based systems does not accurately measure individual load latency tolerance, and the results may be highly dependent on the cache organization. The latency tolerance of references that hit in the cache is not measured, even though it may be quite large.

Our methodology extends this previous work, and is targeted at measuring the latency tolerance of individual load instructions. We rely on the ability to force completion of loads at arbitrary times to ensure the processor is able to continue issuing instructions. Note this approach measures latency tolerance in the context of a processor with constrained resources. Eliminating these constraints would be an interesting study, but is beyond the scope of this paper.

In our scheme, the measurement of load latency is decomposed into the following four steps, which we elaborate on in the remainder of this section:

1. Determining that one or more loads should complete,
2. Determining when the load(s) should complete,
3. Determining which specific load(s) should complete, and
4. Determining how many loads should complete.

3.1 Determining that Loads Should Complete

Our goal is to allow loads to remain outstanding as long as they are not adversely affecting performance. Therefore, to determine if any load(s) should be forced to complete we must first determine if the processor performance is degrading. Recall that the performance of dynamically scheduled processors can degrade because it is unable to execute independent instructions due to limited buffer space, data dependencies, or it executes useless instructions due to incorrect branch prediction. Therefore, we force loads to complete if their results ensure the processor executes useful instructions or enable the processor to sustain reasonable execution rates.

Branch-based Load Completion

Most modern processors predict the outcome of branches and speculatively execute instructions on the predicted path. On a misprediction, all the work done by the processor in speculative mode is useless. Delaying completion of a load on which a branch instruction is dependent can increase the number of mis-speculated instructions executed and therefore degrade performance. Hence, loads on which branches are dependent need to be given priority for early completion. Moreover, it is only mispredicted branches that cause the processor to execute useless instructions. Therefore, it is sufficient to force a load to complete as soon as a mispredicted branch attaches itself to the load's dependency graph.

Performance-based Load Completion

Using branch prediction information to force completion of certain loads ensures the processor is executing useful instructions. However that alone is not enough. Arbitrarily delaying completion of the rest of the loads will aggravate the data dependencies problem and the finite resources problem mentioned in the previous section. This could prevent the processor from sustaining a reasonable level of performance.

To decide if loads should complete because of processor performance, we can monitor one of two standard processor performance metrics: instruction issue rate or functional unit utilization. When the processor performance drops, we complete loads freeing up dependent instructions as well as buffer space. In order to attain high IPCs we do not delay load completion until the processor actually comes to a stand still. Rather, we complete loads as soon as the number of instructions issued or the number of computational units that are busy drops below a tunable threshold.

Loads can also be forced to complete when there is a system call. However, for our benchmarks there are very few system calls, therefore we do not discuss this case further in this paper.

3.2 Determining Which Loads to Complete

Once it is determined that some load(s) must complete, we need to decide which specific load to complete. In the case of mispredicted branches, clearly the load on which the branch is dependent must be completed to ensure the execution of useful instructions. In contrast, we have complete freedom to choose any load for completion when the issue rate or functional unit utilization decreases. We investigate two policies: fifo and dependence graph depth. The fifo policy simply forces the longest outstanding load

to complete. The second policy tracks the depth of a load's dependence graph in cycles. The load with the largest value is chosen for completion, since delaying it can occupy resources for an extensive period of time.

3.3 Determining When Should Loads Complete

Having established which loads to complete, the next step is to determine when (i.e., in which cycle) they must complete. To minimize execution of useless instructions due to mispredicted branches, we must complete the appropriate load such that the entire dependence chain between the load and the branch completes execution before the branch. This requires the load to complete many cycles before we actually detect the mispredicted branch. Section 4 describes how our simulations accomplish this. If the load is forced to complete because of issue rate or functional unit utilization, we could naively complete it in the same cycle that we detected the degradation in performance. However, this may not provide enough time for the pipeline to fill up with ready instructions, and we may want the load to complete earlier. Therefore, we use a tunable threshold for load *precompletion* time to study the effect of pipeline fill-up time on load latency tolerance.

3.4 Determining How Many Loads to Complete

Finally, in order to obtain an instruction issue rate or functional unit utilization above the set threshold, we may need to complete more than one load in a given cycle. An important parameter in this scenario is the limit on the number of loads that may complete. We study this by limiting the number of loads that can complete in a single cycle to one, two, or four.

4 Experimental Methodology

To perform our measurements we modified SimpleScalar [2], which models a dynamically scheduled processor using a Register Update Unit (RUU) and a Load/Store Queue (LSQ) [16]. The processor pipeline stages are:

Fetch: Fetch instructions from the program instruction stream.

Dispatch: Decode instructions, allocate RUU, LSQ entries.

Issue/Execute: Execute ready instructions if the required functional units are available.

Writeback: Supply the results of the operation to dependent instructions.

Commit: Commit results to the register file in program order, free RUU and LSQ entries.

Our baseline processor is an 8-issue machine with 8 integer adders, 4 integer multiply/divide units, 8 floating point adders, 4 floating point multiply/divide units, and 8 cache ports. We assume 256 RUU entries and 128 LSQ entries, a 2-level branch predictor with a total of 8192 entries, and that all stores complete in a single cycle.

When necessary we assume a base two level cache configuration using a 32KB direct-mapped L1, with 32 byte blocks and 8 ports. The L2 is 1MB direct-mapped with 64 byte blocks, a single port and 8 cycles to satisfy an L1 miss. Both caches support up to 16 outstanding misses, are fetch-on-write writeback, and have a 24 entry write-back buffer with a high watermark of 12. Contention is modeled in all parts of the memory system.

Many of the load completion policies outlined in the previous section decouple detecting that a load must complete from determining when the load should complete. Therefore, it is possible for our scheme to determine that a load should have completed even before we detect that it should complete. Recall the scenario where we detect that a load should complete when a branch is dispatched and attaches itself to the dependence graph of the load. Assume this detection occurs at cycle t and there are d cycles worth of instructions in the dependence chain from the load to the branch. To minimize execution of useless instructions, we determine that the load should complete at cycle (t-d), d cycles before we even establish the load should complete.

To support this type of analysis, we added rollback capabilities to our simulator. This allows us to look ahead to compute load completion time, rollback the processor, and then restart execution using the predetermined load latency. The replayed execution may itself incur rollbacks. This technique ensures the processor instruction schedule is determined by the measured latency values. Supporting rollback requires logging all processor state at the end of each simulated cycle. We limit the maximum number of cycles a load can be outstanding to 32 and therefore a single load can cause the processor to rollback a maximum of 32 cycles.

Simulating a detailed out-of-order processor takes an enormous amount of time, and the rollback capabilities we added only increase simulation time. Therefore, we also modified SimpleScalar to support sampling. Our sampling technique alternates between a detailed out-of-order simulator and a faster functional simulator that also maintains the contents of the memory hierarchy.

Finally, to evaluate the effectiveness of traditional memory hierarchies at capturing latency tolerance, we simulate a two-level memory hierarchy, as described above, in the same execution as the latency tolerance analysis. This enables comparison between the measured latency tolerance and where in the conventional memory hierarchy the request is satisfied. The load timing is dictated by the latency tolerance measurements, and we simply track the contents of the memory hierarchy. Because of the different processor schedule, there may be some inac-

curacies on the contents of the caches compared to an execution with load latency dictated by the conventional caches. However, we believe our approach is sufficient for this study.

The following section presents our analysis using a subset of the SPEC95 benchmarks: `compress`, `gcc`, `li`, `vortex`, `hydro2d`, `swim`, `tomcatv`, and `wave`. The benchmarks are all compiled using the version of `gcc` provided with SimpleScalar and with optimization -O2. We run each benchmark operating on its *reference* data set until 10 billion instructions commit using 1% sampling. This sampling ratio produces IPC values within 5% of complete simulations.

5 Experimental Results

This section presents our simulation results. We begin by examining the performance of our benchmarks for different memory systems. This is followed by analysis of the effects of branch prediction on latency tolerance. We then analyze the effects of varying how to determine that loads should complete, how to select loads to complete, how to compute the time that loads should complete, and how many loads are completed. We finish by examining various processor configurations and investigating the match between the latencies incurred in conventional multi-level memory hierarchies and the program's measured latency tolerance.

5.1 Fixed Latency Memory Systems

One approach to obtain information on the amount of latency tolerance in a system is to evaluate its performance for various memory system delays. We performed this experiment by examining memory systems ranging from simple fixed cost memory accesses with no contention to detailed memory hierarchies with contention accurately modeled at all levels. The fixed cost memory systems assume all loads take the same amount of time, we examined 1, 8, and 32 cycle memory accesses. The detailed two-level memory hierarchies assume the base 1MB second level cache, but vary the first-level configuration and the second-level miss penalty. Specifically, we simulated a direct-mapped and two-way set-associative 32KB L1 cache with a 32 cycle memory latency (memlat32-{dm,2way}32k) and a direct-mapped 32KB L1 with a 64 cycle memory latency (memlat64-dm32k).

Figure 1 shows the performance of our benchmarks in terms of committed instructions per cycle (IPC) for the above memory system configurations. We make several observations from these results. First, for three of the integer benchmarks (`gcc`, `li`, and `vortex`) the traditional memory systems achieve IPC values close to the ideal memory system. This is not surprising, given the low miss

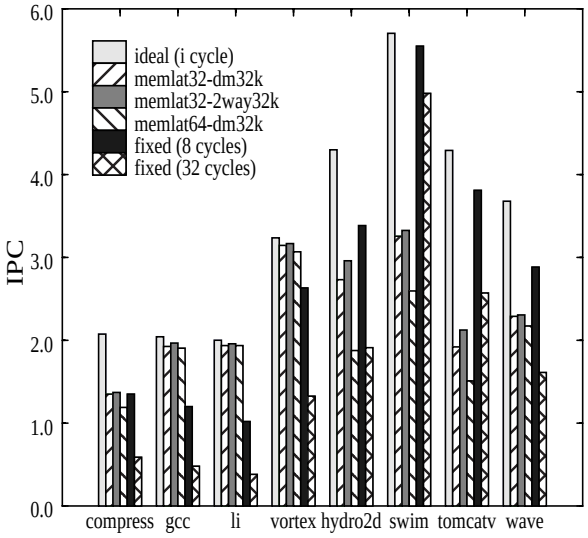

Figure 1: Memory System Configuration and IPC

ratios of these benchmarks, 2%, 1.4%, and 1.5% for `gcc`, `li`, and `vortex`, respectively, for a direct-mapped 32KB L1 cache. The other benchmarks exhibit L1 miss ratios over 4%, thus increasing the discrepancy in performance compared to the ideal memory system. We note that increased associativity has little effect on overall IPC, and that increasing the L2 miss penalty (memlat64-dm32k) dramatically reduces the performance of three floating point benchmarks (`hydro2d`, `swim`, `tomcatv`), while all other benchmarks exhibit a small reduction in IPC.

Another observation from the data in Figure 1 is that the performance of all benchmarks decreases as we increase the latency for all memory accesses (ideal, fixed 8 cycles, fixed 32 cycles). The integer programs are especially sensitive to the increases in fixed cost memory delays, and their IPC values drop below the traditional memory system when all memory accesses take 8 cycles. In contrast, the floating point codes show less sensitivity to a fixed cost delay of 8 cycles. Further increases in memory latency continue to decrease the IPC for all programs. However, we point out the results of swim that show only moderate reduction in IPC even when all memory accesses take 32 cycles. This performance is dramatically higher than the detailed two-level memory hierarchy mainly because of contention within the memory hierarchy.

The above analysis of fixed cost memory accesses provides some insight into latency tolerance. However, it is an all or nothing approach where every load has the same cost, and is suitable only for specific memory system designs. The uniform cost model doesn't exist in multi-level memory hierarchies where some loads can be satisfied faster than others. Therefore, as described in Section 3, our methodology is targeted at measuring the

latency tolerance of individual load instructions. The remainder of this section presents our results.

5.2 Determining that Loads Must Complete

This section investigates the policies for determining when loads must complete in order to sustain performance comparable to an ideal memory system. We begin by examining how branch prediction affects load latency tolerance. This is followed by analysis of instruction issue rate and functional unit utilization as metrics for determining load completion.

Branch Prediction and Load Latency Tolerance

We compare perfect branch prediction to our base two-level predictor using various policies for determining that loads should complete. For the two-level branch predictor, the first policy always forces loads to complete if any branch attaches itself to the load's dependence chain (2-lev, all). The next policy is similar, except it forces a load to complete only if the branch is mispredicted (2-lev, mispred).[1] Finally, for both the two-level and perfect branch predictor we evaluate a policy that does not use any branch information to force completion of loads (2-lev, none and perfect, none). In all of these simulations we make the following assumptions, loads not forced to complete by a branch are completed according to an instruction issue threshold of four instructions per cycle, up to four loads can complete per cycle, which load to complete is determined by the dependence graph depth, and we assume a precompletion time of two cycles. We evaluate these parameters later in this section. For comparison, we also simulate the ideal memory system for both the two-level predictor (2-lev ideal) and perfect prediction (perfect ideal).

Throughout this section we present our results in two parts: IPC and latency tolerance. IPC results are presented like those in Figure 1. We present latency tolerance in terms of the fraction of loads that must complete in a specific number of cycles. Loads that must complete in a small number of cycles, do not exhibit latency tolerance and loads that can take many cycles to complete do exhibit tolerance. Loads are forced to complete according to the appropriate policy, or if they've been outstanding for 32 cycles.

Figure 2 shows the effects of branch prediction on IPC, while Figure 3 shows the corresponding latency tolerance values. From Figure 2, we see that the 2-level predictor policies that exploit branch information to force load completion, meet our goal of IPC close to an ideal memory system. Furthermore, we see that using only mispredicted branches produces similar IPC values as forcing loads to complete for all branches, and that ignoring branch information entirely dramatically reduces the integer programs' IPC values. We also see the expected result that perfect branch prediction dramatically increases performance for the integer codes. The two-level predictor achieves only 88% accuracy for `compress`, 81% for `gcc`, 86% for `li`, and 89% for `vortex`. The floating point codes exhibit somewhat higher prediction rates (`hydro2d` 99%, `swim` 99%, `tomcatv` 92%, `wave` 90%). More sophisticated branch predictors may produce higher accuracies, hence increased IPC rates.

For the integer benchmarks `compress`, `gcc` and `li`, between 20% and 25% of the loads are completed based on mispredicted branch information, 7% for `vortex` and less than 2% for the floating point benchmarks. With load

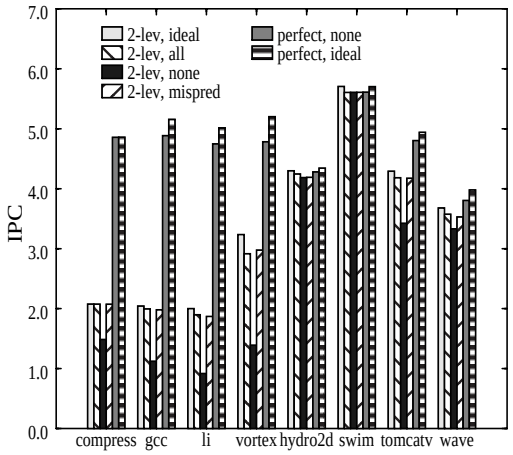

Figure 2: Branch Prediction and IPC.

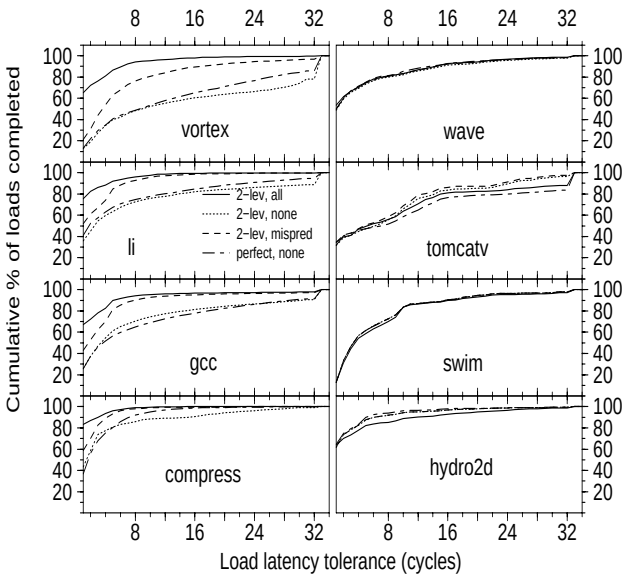

Figure 3: Branch Prediction and Latency Tolerance

1. This is possible to simulate because in SimpleScalar we can determine very early in the simulation cycle if a branch is mispredicted.

completion based on mispredicted branches enabled, we see a considerable reduction in the average dispatch-issue delay for branches (time for the operands of the branch to become available) for the integer benchmarks. Also, the number of speculative instructions executed drops by up to 61% for the integer benchmarks. The effect on the floating point benchmarks is considerably less.

From Figure 3 we see that loads do exhibit variation in completion delays, and there is significant variation among benchmarks. Between 13% and 62% of loads need to complete in one cycle, while 58% to 98% of the loads need to complete within eight cycles. Furthermore, the floating point programs exhibit very little variation in load latency for the various branch-based load completion schemes. In contrast, the integer programs are sensitive to these factors. In particular, differentiating mispredicted branches from accurately predicted branches produces noticeable improvements in load latency tolerance without significant changes in IPC. Ignoring branches altogether yields high latency tolerance, but the IPC values are too low. The final observation from these results is that improvements in branch prediction will increase the amount of latency tolerance for the integer programs, as indicated by the increases seen for perfect branch prediction.

Processor Performance and Load Latency Tolerance

The second source of information for determining if loads should complete is processor performance. Here, we examine the instruction issue rate and functional unit utilization as metrics for determining that loads should complete. We assume that mispredicted branches force completion of loads, precompletion time is 2 cycles, and up to four loads can complete per cycle.

Figure 4 shows the effect of issue rate thresholds of 1 (nis1), 2 (nis2) and 4 (nis4), and a functional unit utilization threshold of 4 (fub4) on instructions per cycle. Whenever the processor issue rate (or number of busy functional units in the case of fub4) drops below this threshold, we force loads to complete. For comparison, we include the IPC values for the traditional memory system (memlat32-dm32k) and the fixed 8 cycle memory system. From this data we see that functional unit utilization produces slightly higher IPC values than instruction issue rate (fub4 vs. nis4). The simulations also reveal that decreasing the instruction issue rate threshold produces a commensurate decrease in IPC. We note that for all but three of the integer benchmarks, IPC values are still higher than the traditional two-level memory system even when the threshold is one instruction per cycle. As mentioned previously, the three integer programs have low L1 miss rates, and they achieve near ideal performance. We also note that for `swim`, functional unit utilization actually achieves higher

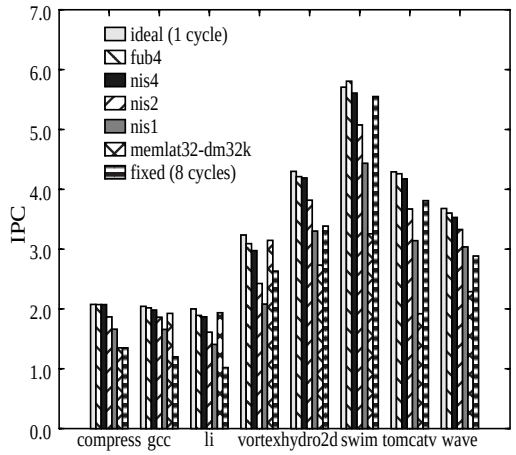

Figure 4: Performance-based completion and IPC

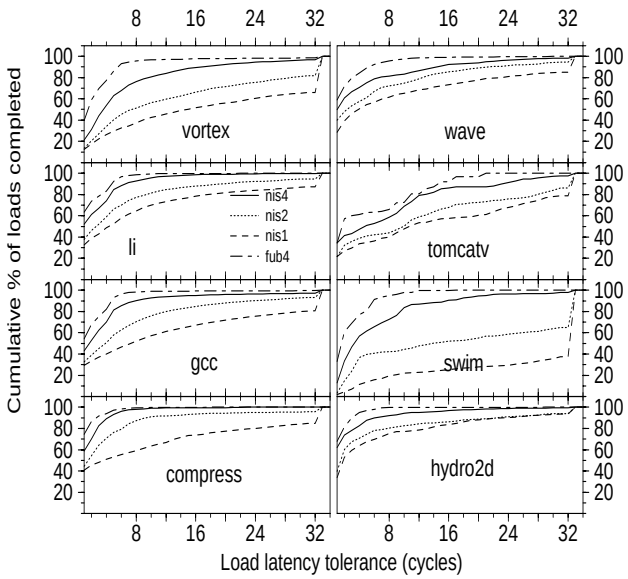

Figure 5: Performance-based completion and Latency Tolerance

IPC than the ideal memory system because of the difference in the processor instruction issue schedule.

Figure 5 shows the corresponding latency tolerance for the various issue rate thresholds. The first observation is that although functional unit utilization (fub4) has a slight performance advantage, it produces much lower latency tolerance than the instruction issue rate metric (nis4). We also observe that decreasing the issue rate threshold can dramatically increase the latency tolerance. This matches our intuition that if the processor is consuming data at a lower rate, it can take longer for the data to arrive. However, the cost of this increased latency tolerance is reduction in IPC. A four instruction per cycle threshold produces IPC values within 8% of the ideal memory system, whereas a threshold of one instruction per cycle pro-

duces IPC values up to 35% lower than ideal. Therefore, we do not consider thresholds of one or two further in this paper. Similarly, we omit further discussion of functional unit utilization since the decreased latency tolerance more than offsets the marginal increase in IPC.

5.3 Determining Which Loads to Complete

Now that we've determined that either mispredicted branches attaching to a load's dependence graph or the instruction issue rate falling below 4 should force load completion, we focus on identifying which outstanding load to complete. Completing the load at the head of the LSQ (fifo) can prevent processor stalls due to the RUU/ LSQ being full and thus help alleviate the finite resource problem. On the other hand, completing the load with the maximum depth (in cycles) of dependent instructions (dg) will help tackle the data dependency problem by freeing up the most dependent instructions and thereby keep the processor maximally utilized

Figure 6 shows the effect of the load selection policy on instructions per cycle and Figure 7 shows the corresponding latency tolerance values. The figures show that both the fifo and dg load selection policies produce almost identical IPC numbers. However completing loads based on the dependence graph depth increases the latency tolerance of loads for the floating point benchmarks. These results provide further evidence of the variation in load latency tolerance, and indicate that completing loads in program order is not necessarily the "best" schedule for exploiting latency tolerance.

5.4 Determining When to Complete Loads

Having decided to complete the loads with the maximum depth of dependent instructions, we proceed to investigate when such loads should be completed. The load completion time controls the amount of time available for the pipeline to fill up with ready instructions. We study the effect of completing loads the same cycle as detecting performance degradation (pipeline fill-up time of zero - fut0), one cycle earlier (fut1) and two cycles earlier (fut2) on load latency tolerance. Note this only applies to loads not forced to complete by a mispredicted branch. From Figure 8,we see that IPC goes down for all benchmarks except `compress`, `gcc` and `li` as we decrease the pipeline fill-up time. These three benchmarks have a significant number of loads completed due to mispredicted branches. Hence fill up time has less impact. `Swim` shows the highest degradation in IPC, going down from within 3% of ideal for fut2 to within 10% of ideal for fut0. Looking at the corresponding latency tolerance graphs in Figure 9, latency tolerance generally increases as we decrease the fill up time. These results match our expecta-

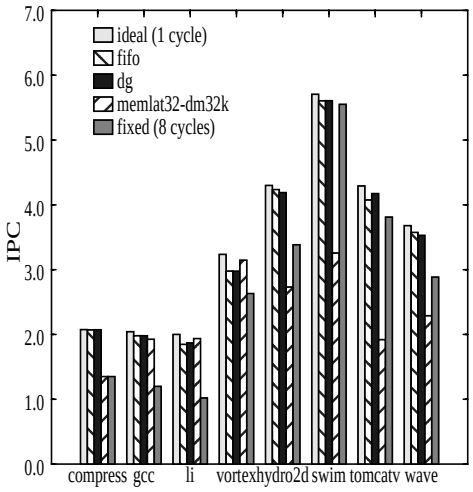

Figure 6: Load Selection and IPC

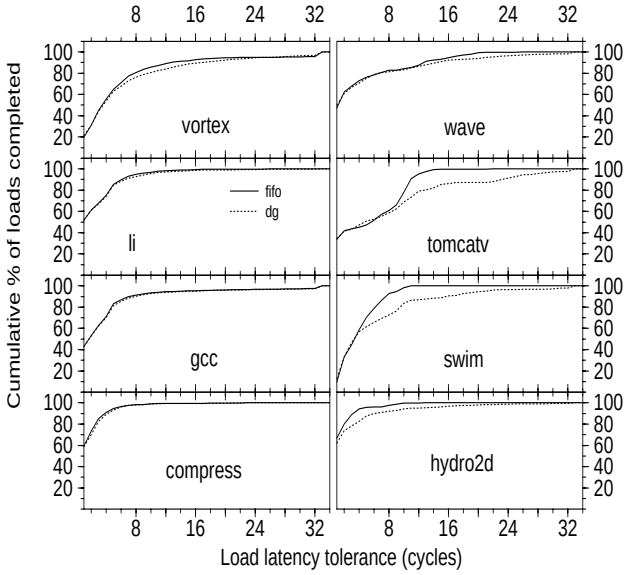

Figure 7: Load Selection and Latency Tolerance

tions, completing loads earlier obviously decreases their latency tolerance. Furthermore, the processor requires some recovery time as results propagate down the dependence graph and a sufficient number of instructions become ready to execute. Using a pipeline fill-up time of 2 cycles (fut2) produces the best combination of IPC and latency tolerance numbers.

5.5 Limiting the Number of Completed Loads

Finally, achieving IPCs close to that of an ideal memory system will likely require completing more than one load per cycle. Keeping all other parameters fixed, we examine limits of one (nl1), two (nl2) and four (nl4) on the number of loads that can complete in a single cycle. Figure 10 shows the impact of these limits on IPC and

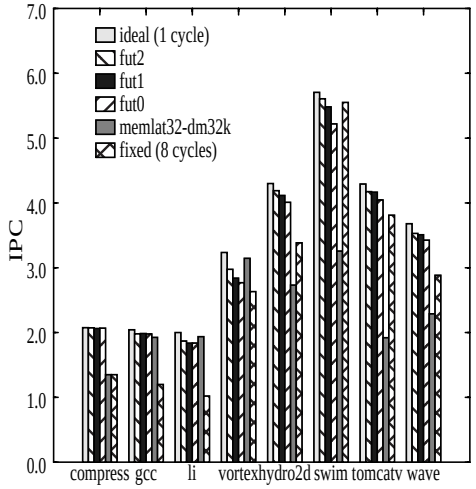

Figure 8: Completion Time and IPC

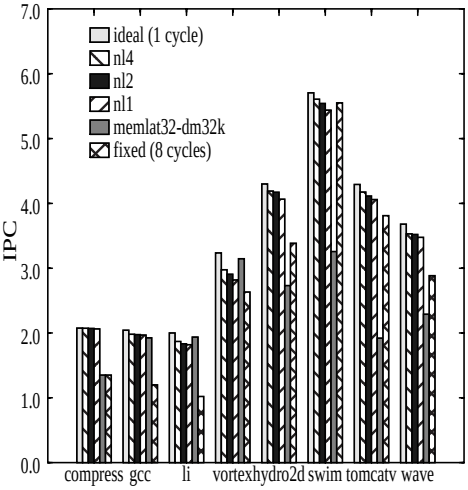

Figure 10: Loads Completed and IPC

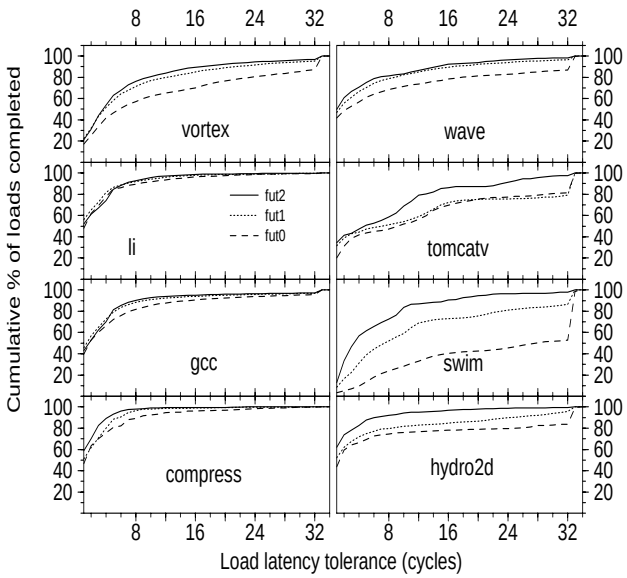

Figure 9: Completion Time and Latency Tolerance

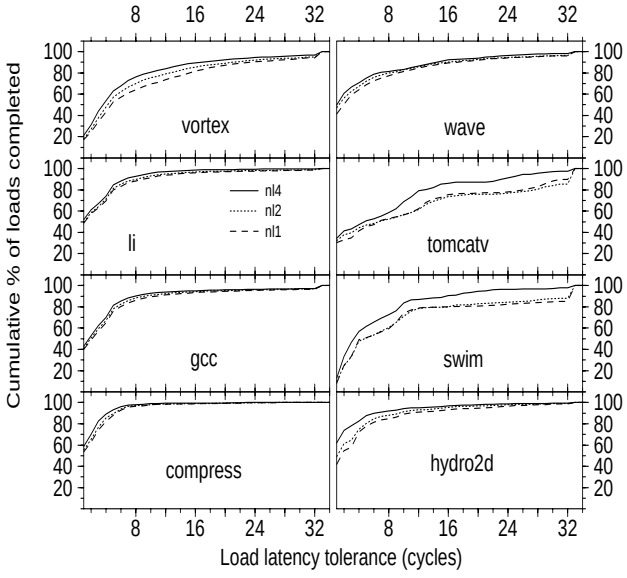

Figure 11: Loads Completed and Latency Tolerance

Figure 11 shows the corresponding latency tolerance numbers. The overall trend we observe from this data is that, as we increase the limit on the number of loads that can complete in a cycle from 1 to 4, IPC increases and the latency tolerance decreases. This is in line with our expectations, since a lower limit causes some loads to complete later than they should according to our policies, which causes a decrease in IPC.

5.6 Effects of Processor Architecture

To evaluate the impact of various microarchitectural changes on our measurements, we evaluated a configuration with 128 RUU entries and 64 LSQ entries, and a four issue processor for both the 128/64 and 256/128 RUU/LSQ configurations. In the case of the four issue processor, we used an issue rate threshold of three instructions per cycle and up to three loads can complete in a single cycle. Figure 12 shows the effect of various processor configurations on IPC and Figure 13 shows the corresponding latency tolerance numbers. The graphs are labeled according to issue-width/RUU entries/LSQ entries.

The first observation from this data is that the issue width has a much larger impact on IPC than buffer space does. We note that all the IPC values shown are within 11% of the corresponding ideal memory system. Also, we see that the IPC values are mostly independent of the number of RUU/LSQ entries. However, the situation is very different with respect to the amount of latency tolerance. From Figure 13, we see that latency tolerance increases when either the issue width decreases or the RUU/LSQ entries increases. The floating point programs exhibit

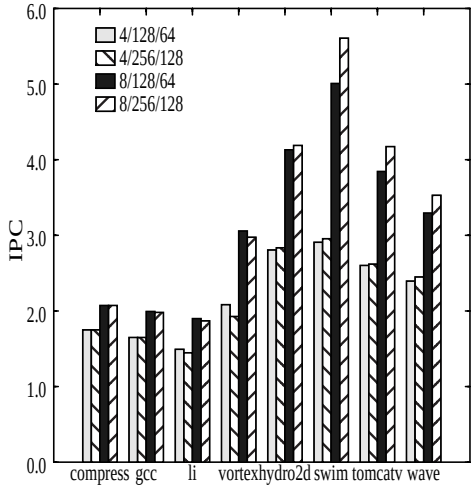

Figure 12: Processor Architecture and IPC (Issue-width/RUU-size/LSQ-size)

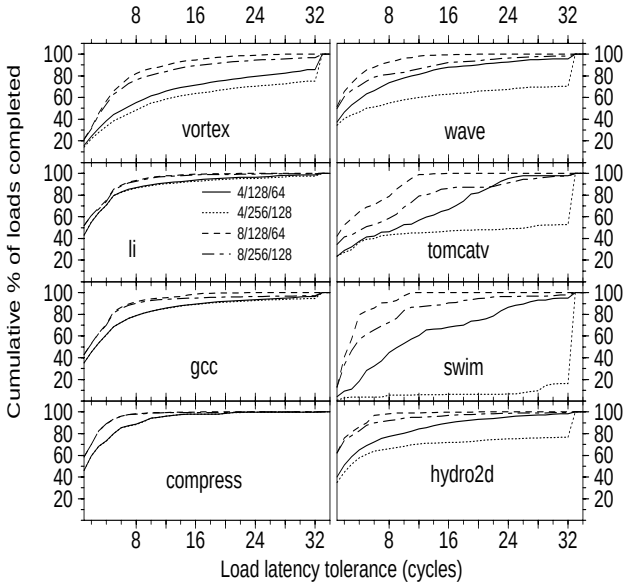

Figure 13: Processor Architecture and Latency Tolerance (Issue-width/RUU-size/LSQ-size)

larger increases than the integer programs. The most striking change is for `swim` with the 4/256/128 configuration, nearly all its memory references can complete in the 32 cycle limit.

5.7 Traditional Memory Hierarchies

The results presented thus far indicate that programs do exhibit variations in load latency tolerance. We now evaluate how well traditional memory systems meet an application's inherent latency demands. To determine this, we track which level in a traditional memory hierarchy satisfies each load and increment a counter for the corresponding measured latency tolerance. This produces a histogram for each level in the memory hierarchy, and allows us to evaluate the memory hierarchy's effectiveness with varying access times for each level. For example, if the L2 access time is 8 cycles, then to avoid performance degradation, loads with measured latency tolerance less than 8 cycles should be satisfied by the L1 cache. Similarly, loads with measured latency greater than or equal to 8 cycles, but less than main memory access time, could be satisfied by the L2 cache. Clearly, we can perform this computation for arbitrary access times. Furthermore, we can track the discrepancy between where a load should be satisfied and where it is actually satisfied in the memory hierarchy.

We performed this analysis on 8KB, 16KB, and 32KB, direct-mapped and two-way set-associative L1 caches with 32-byte blocks, using the base L2 cache configuration (1MB, 64-byte blocks). For brevity, we report results only for the direct-mapped caches and an L2 access time of 8 cycles on the 8-issue processor with 256 RUU entries and 128 LSQ entries. Table 1 shows the effectiveness of the traditional two-level memory hierarchies at capturing latency tolerance. The top row for each benchmark in the table indicates the percentage of loads satisfied by a particular level in the memory hierarchy with measured latency tolerance less than 8 cycles. Similarly, the bottom row for each benchmark corresponds to loads with measured latency greater than or equal to 8 cycles. The levels of the memory hierarchy are the load/store queue (LSQ), L1 cache, L2 cache, and main memory.

We focus our discussion on the L1 and L2 caches. In particular, the number of low latency loads (< 8 cycles) not satisfied by the L1 cache and the number of high latency loads (>= 8 cycles) satisfied by the L1 indicates the mismatch between the applications latency demands and the latency incurred in the memory hierarchy. From these results we see that some benchmarks exhibit significant discrepancy between their latency demands and the latency that's incurred in a real memory hierarchy. Consider the 16KB cache for `swim`, 16% of loads require low latency but are satisfied by the L2 cache, whereas 29% of loads have enough latency tolerance and are L1 cache hits. Ideally, those references should be swapped, with the L1 cache satisfying the low latency loads and the L2 satisfying the high latency loads

`Compress` is another striking example, with 16% to 22% of its loads requiring low latency but missing in the L1 cache. However, we note that `compress` has very few high latency loads for this processor configuration. In contrast, for `swim` 2% to 36% of loads require low latency yet miss in the L1 cache, whereas 13% to 37% of loads are high latency and hit in the L1. Finally, for the floating point benchmarks a noticeable fraction (1%-3%) of loads

Benchmark	Latency	Where the Load is Satisfied in a Traditional Memory System											
		LSQ			L1 Cache			L2 Cache			Memory		
		8K	16K	32K	8K	16K	32K	8K	16K	32K	8K	16K	32K
compress	< 8	11%	11%	11%	65%	68%	71%	22%	19%	16%	0	0	0
	>= 8	0	0	0	2%	2%	2%	1%	1%	0	0	0	0
gcc	< 8	7%	7%	7%	75%	77%	73%	5%	3%	2%	0	0	0
	>= 8	1%	1%	2%	10%	10%	16%	1%	0	0	0	0	0
li	< 8	9%	9%	9%	80%	81%	81%	3%	2%	2%	0	0	0
	>= 8	2%	2%	2%	7%	7%	7%	0	0	0	0	0	0
vortex	< 8	17%	17%	17%	52%	53%	54%	4%	3%	2%	0	0	0
	>= 8	5%	5%	5%	21%	21%	21%	1%	1%	0	0	0	0
hydro2d	< 8	22%	22%	22%	60%	61%	62%	6%	5%	4%	3%	3%	3%
	>= 8	0	0	0	7%	8%	8%	1%	1%	1%	0	0	0
swim	< 8	5%	5%	5%	26%	45%	59%	36%	**16%**	2%	2%	2%	3%
	>= 8	0	0	0	13%	**29%**	37%	18%	8%	1%	1%	1%	1%
tomcatv	< 8	11%	11%	11%	20%	31%	39%	22%	11%	2%	3%	3%	4%
	>= 8	0	0	0	17%	31%	36%	21%	7%	3%	6%	6%	6%
wave	< 8	26%	26%	26%	37%	44%	48%	17%	10%	6%	1%	1%	1%
	>= 8	4%	4%	4%	8%	12%	14%	7%	3%	1%	0	0	0

Table 1: Effectiveness of Traditional Memory System at Capturing Latency Tolerance

require low latency but are satisfied by main memory. As the disparity between processor cycle time and main memory access time increase, even this small fraction of references can dramatically reduce overall performance.

The mismatch between an application's latency demands and the actual latency is dependent on the performance of the real memory hierarchy. In general, we see that reducing the L1 cache size increases the discrepancy. The floating point benchmarks show dramatic increases as the cache size is reduced. `Swim` and `tomcatv` go from 2% low latency L1 load misses in the 32KB cache to 36% and 22%, respectively, in the 8KB cache. Although not shown, reducing the boundary from 8 cycles to 6 cycles can increase the number of low latency load misses in the L1 cache. Also, our results (not shown) indicate that increasing the L1 associativity has very little effect on the match between the application's latency demands and the memory hierarchy's performance, when compared to the direct-mapped caches.

6 Conclusion

This paper explores latency tolerance in dynamically scheduled processors. Our two primary contributions are a quantitative evaluation of applications' inherent latency tolerance in dynamically scheduled processors, and analysis of how well a conventional memory hierarchy meets the application's latency demands. We compute latency tolerance by measuring the number of cycles a load could take to complete without adversely affecting performance compared to an ideal memory system where all loads complete in one cycle.

Our measurements show that load latency is a function of the number and type of dependent instructions. In particular, mispredicted branches have a significant impact on

measured latency tolerance for the integer benchmarks. We also observe that most programs do exhibit some latency tolerance, and still obtain IPC values comparable to an ideal memory system. Our results show that between 1% and 62% of loads must complete in one cycle, and between 5% and 98% must complete within 8 cycles, depending on processor configuration.

We show that for some benchmarks, a significant number of loads could be satisfied in latencies on the order of second level cache access times, while others must be satisfied by the first level cache. Unfortunately, this discrepancy in latency tolerance is ignored by conventional memory hierarchies that always fetch data into the primary cache. We plan to investigate methods for utilizing latency tolerance information in memory hierarchy management. Prefetching [11] is clearly one avenue for exploiting this information. Alternatively, we could place data in the memory hierarchy according to the corresponding load's latency tolerance and bypass higher levels of the memory hierarchy [1,7,20], or prioritize requests in a system that supports multiple outstanding misses.

7 Acknowledgments

We would like to thank Chia-Lin Yang, Mithuna Thottethodi, Robert Wagner, and the anonymous referees for their helpful feedback on earlier versions of this paper.

8 References

[1] Santosh G. Abraham, Rabin A. Sugumar, Daniel Windheiser, B. R. Rau, and Rajiv Gupta. Predictability of Load/ Store Instruction Latencies. *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 139–152, December 1993.

[2] Doug C. Burger, Todd M. Austin, and Steve Bennett. Evaluating Future Microprocessors-the SimpleScalar Tool Set. Technical Report 1308, University of Wisconsin–Madison Computer Sciences Department, July 1996.

[3] S. Dutta and M. Franklin. Block-Level Prediction for Wide-Issue Superscalar Processors. In *Proceedings of 1st International Conference on Algorithms and Architectures for Parallel Processing*, volume 1, pages 143–152, 1995.

[4] Harry Dwyer and H. C. Torng. An Out-of-Order Superscalar Processor with Speculative Execution and Fast, Precise Interrupts. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 272–281, 1992.

[5] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2 edition, 1995.

[6] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. "The superblock: An effective structure for VLIW and superscalar compilation. Technical report, University of Illinois, Urbana, IL Center for Reliable and High-Performance Computing, February 1992.

[7] Teresa L. Johnson and Wen mei W. Hwu. Run-time Adaptive Cache Hierarchy Management via Reference Analysis. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, page to appear, June 1997.

[8] Lizyamma Kurian, Paul T. Hulina, and Lee D. Coraor. Memory Latency Effects in Decoupled Architectures with a Single Data Memory Module. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 236–245, 1992.

[9] Monica S. Lam and Robert P. Wilson. Limits of Control Flow on Parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 46–57, 1992.

[10] K. N. Menezes, S. W. Sathaye, and T. M. Conte. Path prediction for high issue-rate processors. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, pages 178–188, November 1997.

[11] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 62–73, 1992.

[12] Eric Rotenberg, Steve Bennett, and James E. Smith. Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 24–34, Dec 1996.

[13] Stuart Sechrest, Chih-Chieh Lee, and Trevor Mudge. Correlation and Aliasing in Dynamic Branch Predictors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 22–32, 1996.

[14] J.E. Smith. A Study of Branch Prediction Strategies. In *Proceeding of 8th Annual Symposium on Computer Architecture*, pages 135–148, May 1981.

[15] Avinash Sodani and Gurindar S. Sohi. Dynamic Instruction Reuse. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 194–205, June 1997.

[16] Gurindar Sohi. Instruction Issue Logic for High Performance, Interruptable, Multiple Functional Unit, Pipelined Computers. *IEEE Transactions on Computers*, 39(3):349–359, March 1990.

[17] James E. Thornton. Parallel Operation of the Control Data 6600. In *AFIPS Proc. FJCC*, volume 26, pages 33–40, 1964.

[18] R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal*, pages 25–33, January 1967.

[19] Thang Tran and Chuan lin Wu. Limitation of Superscalar Microprocessor Performance. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 33–36, 1992.

[20] Gary Tyson, Matthew Farrens, John Matthews, and Andrew R. Pleszkun. A Modified Approach to Data Cache Management. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, December 1995. to appear.

[21] Tse-Yu Yeh and Yale N. Patt. Alternative Implementations of Two-Level Adaptive Branch Prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 124–134, 1992.