Journal of Instruction Level Parallelism 1 (1999) 1-24 Submitted 5/99; published 10/99

Load Latency Tolerance In Dynamically Scheduled Processors

Srikanth T. Srinivasan SRI@CS.DUKE.EDU
Alvin R. Lebeck ALVY@CS.DUKE.EDU
Department of Computer Science
Duke University
Durham, NC 27708, USA

Abstract

This paper provides a quantitative evaluation 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 a processor with an ideal memory system where all loads complete in one cycle. Our simulations reveal that to produce IPC values within 12% of a processor with an ideal memory system, between 1% and 71% of loads need to be satisfied within a single cycle and that up to 74% can be satisfied in as many as 32 cycles, depending on the benchmark and processor configuration. Load latency tolerance is largely determined by whether a mispredicted branch is in the load's data dependence graph and the depth of the dependence graph. Our results 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 a similar percentage 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 [24,26] 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—with almost no overall performance degradation. To find enough independent instructions, most processors employ sophisticated branch prediction mechanisms [11, 29] and allow speculative execution [19, 12], committing results only when the true outcome of a branch is known.

However, limitations due to finite resources, data dependencies and imperfect branch prediction, render the processor unable to tolerate the latencies of some long latency operations. These operations are likely to degrade processor performance and hence are *critical*. Memory and I/O operations are the only potentially long latency operations whose latency is not fixed. In this paper, we focus only on memory operations. Their latency can be altered by controlling which level in the memory hierarchy holds the requested data. Long latency stores are less of a problem because they do not have other instructions dependent on them, whereas long latency loads can aggravate all three of the above limitations and can reduce performance considerably. In this paper, we concentrate on load instructions.

Even though it is possible for long latency loads to degrade performance, not all of them are equally critical. Figure 1 shows an example program dependence graph with two loads. ld1 has many instructions that are data-dependent on it. These dependent instructions cannot

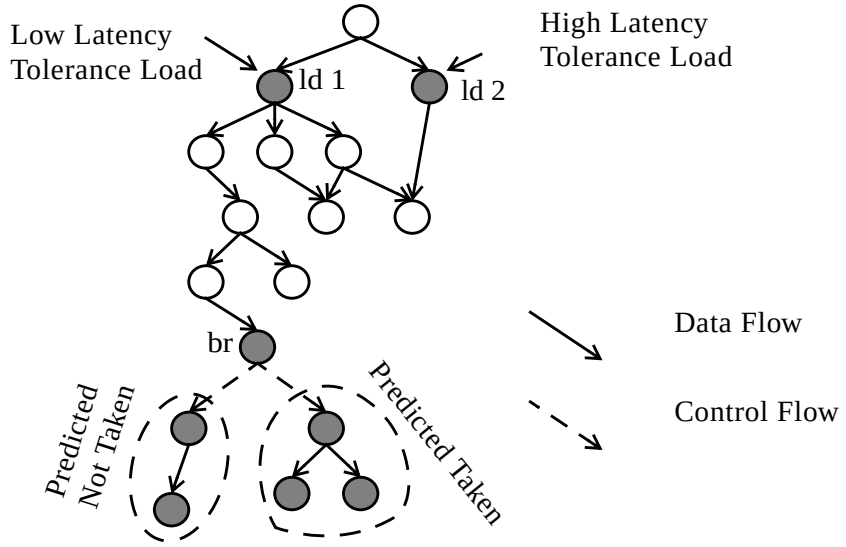


Figure 1: Dependence Graph

begin execution until ld1 completes, and hence can decrease processor utilization. Also, these instructions occupy buffer entries while waiting for their operands to become available. If ld1 has a long latency, the processor could run out of buffer space. Furthermore, ld1 has a branch instruction (br) in its dependence chain. If the branch is incorrectly predicted, then all the work done by the processor executing along the mispredicted path could end up being useless. ld1 should complete quickly to sustain high processor performance. In this scenario, ld1 is said to have *low* latency tolerance. On the other hand, ld2 has only one instruction dependent on it. Even if it takes a long time to complete, the processor might be able to tolerate ld2's latency by executing other independent instructions. In this case, we say that ld2 has a *high* latency tolerance.

The first contribution of this paper is to present a quantitative evaluation of load latency tolerance in a dynamically scheduled processor. Using SimpleScalar [3] we compute individual load instruction latency tolerance by forcing their completion such that the number of instructions committed per cycle (IPC) is comparable to a processor with 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 12% of a processor with an ideal memory system, while yielding noticeable latency tolerance.

Our simulations, on an 8 issue processor that can have up to 256 instructions in flight, show that between 20% and 68% of the loads in our benchmarks need to complete in one cycle and that 67% to 97% must complete in 8 cycles. Reducing the issue width to 4, reduces the number of one cycle loads to between 1% and 45% and the number of 8 cycle loads to 7% to 89%. These results show that many loads could be satisfied with latencies comparable to typical second-level cache hit times.

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 36% 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 and related work. 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 and Related Work

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 architectural register file in program order.

The issue window is filled with instructions from several basic blocks by predicting the direction of conditional branches [5, 6, 13, 17, 20]. Furthermore, the instructions from the predicted path are speculatively executed, under the assumption that branches on the path are correctly predicted. These instructions can not update the architectural state of the processor until the true outcomes of branches are determined. While waiting for 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 one or more 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, program completion could be delayed many cycles because of the time lost due to misprediction detection and correction.

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

2.2 Related Work

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 the latency tolerance of entire programs. 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 determine 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 accounted for, even though it may be quite large. We evaluate the latency tolerance of individual load instructions without being tied down to a specific memory system.

Graph based analyses have been used [2, 9, 15, 21, 25, 28] to study the amount of parallelism available in programs. They work on a static/dynamic execution trace under idealistic assumptions such as unconstrained resources, single cycle operational latencies for functional units, perfect branch prediction and alias analysis. They use the dependence information prior to an instruction to find the optimal schedule for a program. In contrast, we use information about instruction dependencies that follow a load as well as about processor utilization, to determine a load's latency tolerance. Also, we conduct our evaluation on-line on a realistic processor with constrained resources.

3. Computing 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 computed latency tolerance and the latency it incurs in a conventional cache memory hierarchy.

Developing a formal and rigorous definition of load latency tolerance is a complex task. Instead, we use an operational definition of load latency tolerance. 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 computed latencies to reflect a program execution that achieves IPC close to that of a processor with an ideal memory system, where all references complete in one cycle.

We compute the latency tolerance of a load by counting the number of cycles that elapse

1. Some of the results obtained during mis-speculative execution could prefetch useful instruction [16] and data [22] that will be used later.

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.

Our methodology is targeted at determining 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 evaluates latency tolerance in the context of a processor with constrained resources. Extending this study beyond the limits of real hardware may provide interesting insights, but is beyond the scope of this paper.

In our scheme, computing 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 Detecting 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 (instructions on the correct path) or enable the processor to sustain execution rates close to those achievable with a processor with an ideal memory system, where all memory accesses complete in one cycle.

3.1.1 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.

3.1.2 Performance-based Load Completion

Using branch prediction information to force completion of certain loads ensures that loads do not cause the processor to execute useless 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 decrease the execution of useless 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 evaluation we modified SimpleScalar [3], which models a dynamically scheduled processor using a Register Update Unit (RUU) and a Load/Store Queue (LSQ) [23]. 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 (2,2) branch predictor with a total of 8192 entries, and that all stores complete in a single cycle. We use a dynamic memory disambiguation scheme that prevents loads from issuing when any previous store's address is not known, or when the data to be written by a previous store to the same address is not ready. For the sake of simplicity, we ignore advanced memory disambiguation techniques such as store sets [4], as well as load value or address prediction[10].

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 [18]. 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 computed 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 computed latency tolerance and where in the conventional memory hierarchy the request is satisfied. The load timing is dictated by the computed latency tolerance, and we simply track the contents of the memory hierarchy. Because of the different processor schedule, there may be some inaccuracies 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` as well as two other benchmarks: `meteor` and `otter`. 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 detect 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 computed 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 2 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 four of the non-floating-point (non-fp) benchmarks (`gcc`, `li`, `vortex`, and `meteor`) the traditional memory systems achieve IPC values close to the ideal memory system. This is not surprising, given the low miss ratios of these benchmarks, 2%, 1.4%, 1.5%, and 2.8% for `gcc`, `li`, `vortex`, and `meteor` 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 (except `tomcatv`), and that increasing the L2 miss penalty (memlat64-dm32k) dramatically reduces the performance of two floating point benchmarks (`hydro2d`, `swim`), while all other benchmarks exhibit a small reduction in IPC.

Another observation from the data in Figure 2 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 non-fp 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

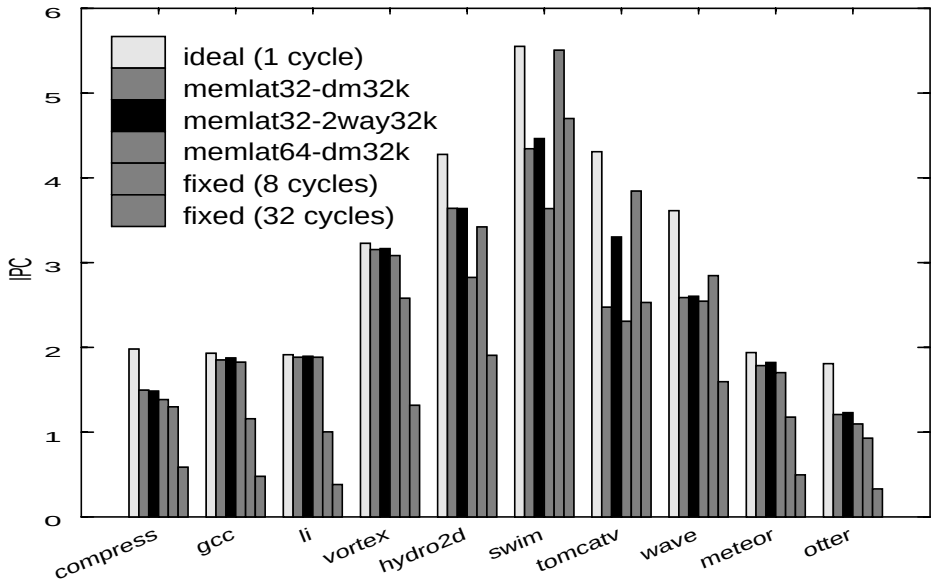


Figure 2: Memory System Configuration and IPC

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 (e.g., L1 to L2 cache bus, L2 to memory bus).

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 Detecting that Loads Must Complete

This section investigates the policies for determining when loads must complete in order to sustain performance comparable to that of a processor with 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.

5.2.1 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).[2] Finally, for both the two-level and perfect branch predictor we evaluate a policy that does not use any branch information to force completion

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

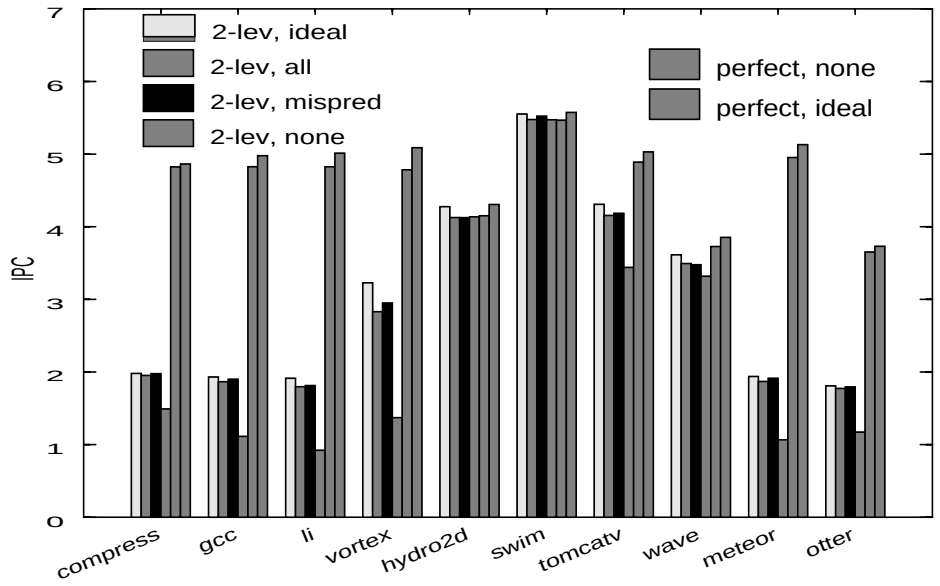


Figure 3: Effects of Branch Prediction on IPC

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 pre-completion 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 2. 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 3 shows the effects of branch prediction on IPC, while Figure 4 shows the corresponding latency tolerance values. From Figure 3, we see that the 2-level predictor policies that exploit branch information to force load completion(2-lev, all and 2-lev, mispred), meet our goal of IPC close to a processor with 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 non-fp programs' IPC values. We also see the expected result that perfect branch prediction dramatically increases performance for the non-fp codes. The two-level predictor achieves only 88% accuracy for `compress`, 81% for `gcc`, 86% for `li`, 89% for `vortex`, 83% for `meteor`, and 89% for `otter`. 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 non-fp benchmarks `compress`, `gcc`, `li`, `meteor`, and `otter`, between 20% and 26% of the loads are completed based on mispredicted branch information, 7% for `vortex` and less than 3% for the floating point benchmarks. With load completion based on

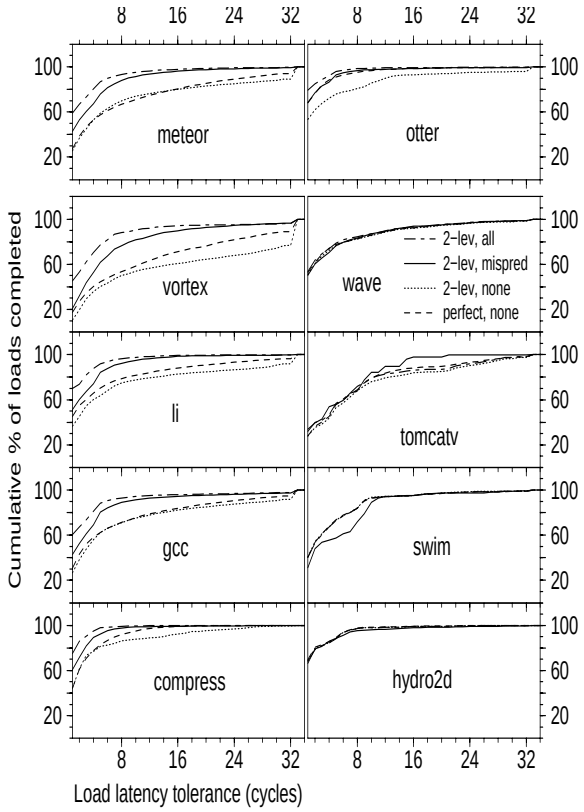


Figure 4: Effects of Branch Prediction on Load Latency Tolerance

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 28% for the non-fp benchmarks. The effect on the floating point benchmarks is considerably less.

From Figure 4 we see that loads do exhibit variation in completion delays, and there is significant variation among benchmarks. Between 10% and 74% of loads need to complete in one cycle, while 50% to 99% 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 non-fp 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.

5.2.2 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

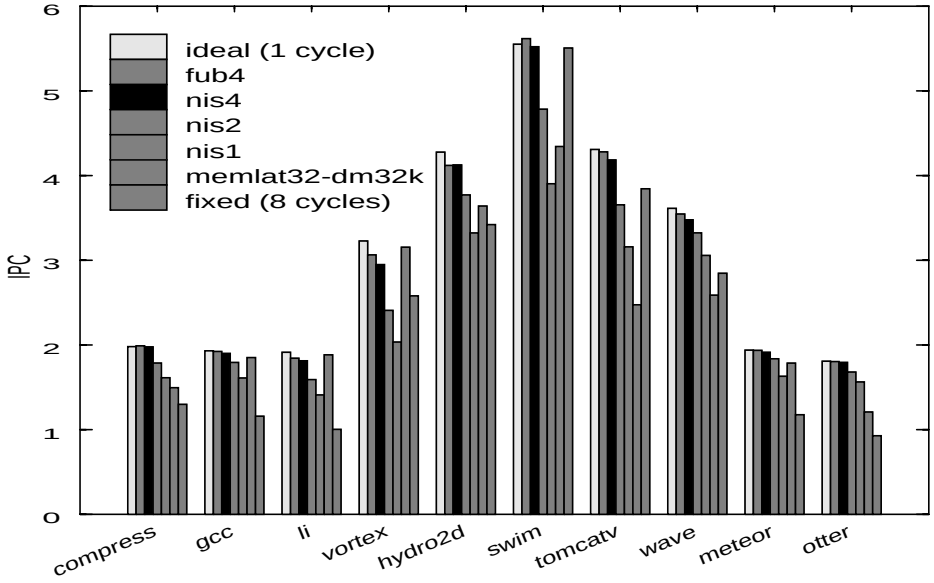


Figure 5: Effects of Performance-based Completion on IPC

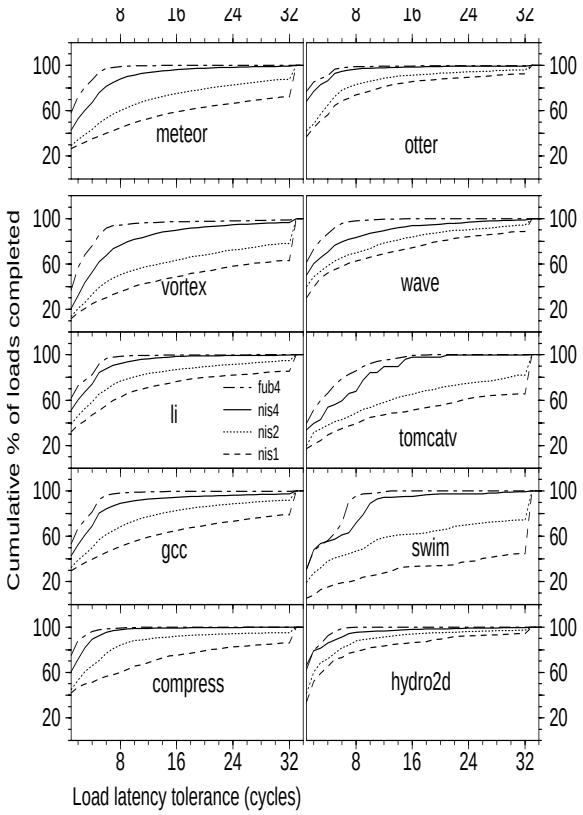


Figure 6: Effects of Performance-based Completion on Load Latency Tolerance

per cycle.

Figure 5 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 pro-

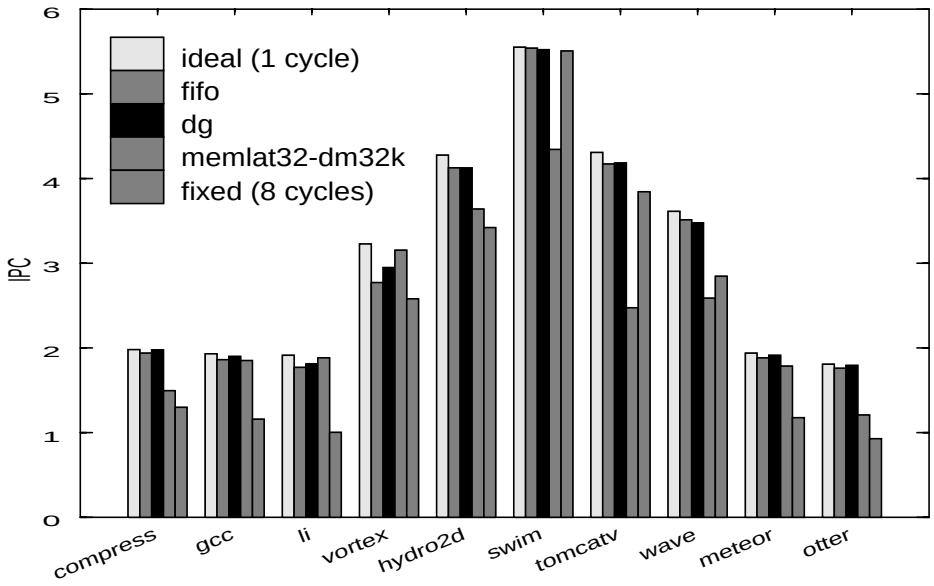


Figure 7: Effects of Load Selection on IPC

cessor 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 the metric 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 four of the non-fp 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 four non-fp programs have low L1 miss rates, and they achieve near ideal performance. We also note that `swim`, using functional unit utilization to force load completion, actually achieves higher IPC than a processor with an ideal memory system because of the difference in the dynamic instruction issue schedule.

Figure 6 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 9% of that of a processor with an ideal memory system, whereas a threshold of one instruction per cycle produces IPC values up to 58% 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 four should force load comple-

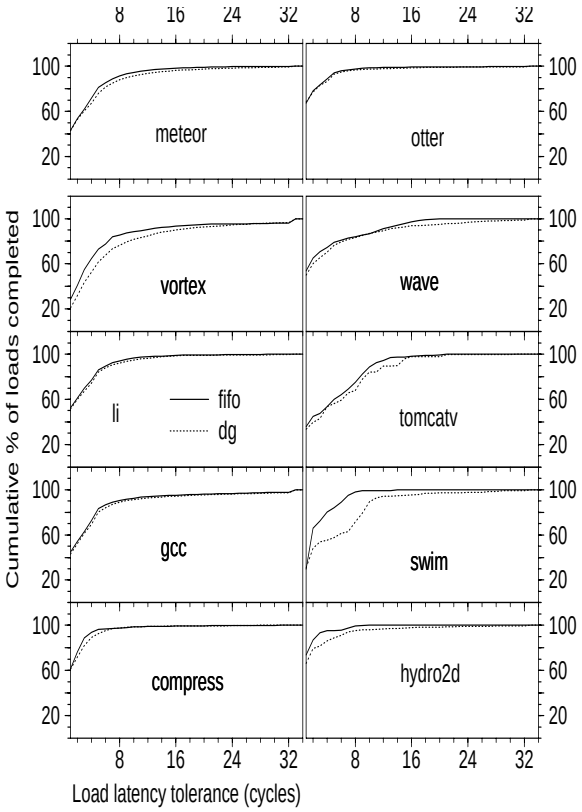


Figure 8: Effects of Load Selection on Load Latency Tolerance

tion, 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 7 shows the effect of the load selection policy on instructions per cycle and Figure 8 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. Therefore, dg is a better choice since it meets our goals of maximizing the time a load can remain outstanding while maintaining high performance. 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 load 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

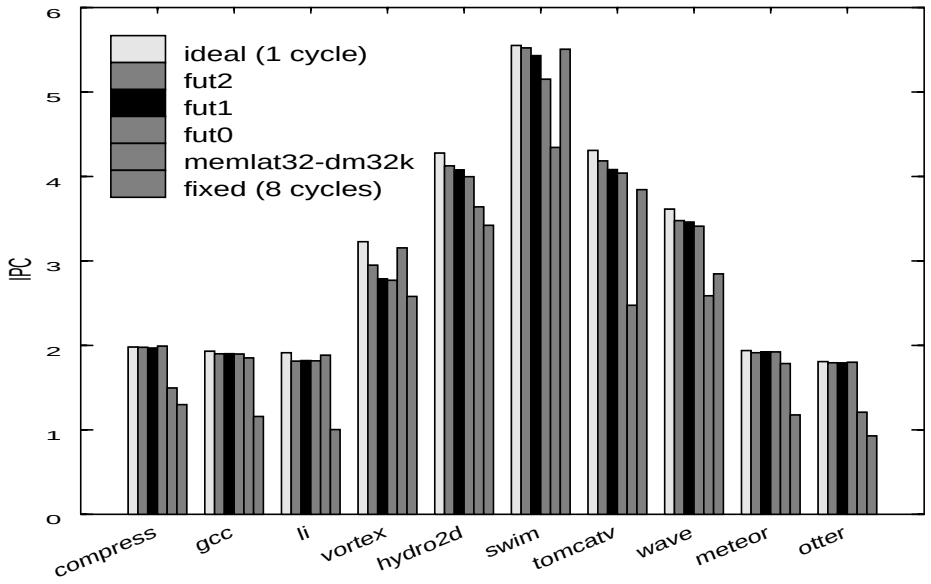


Figure 9: Effects of Completion Time on IPC

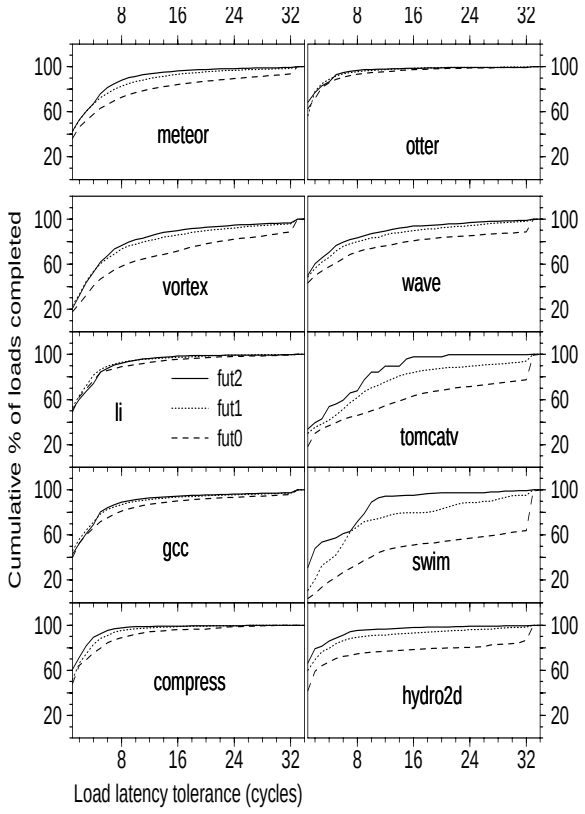


Figure 10: Effects of Completion Time on Load Latency Tolerance

(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 9 we see that IPC decreases as the fill up time increases for the

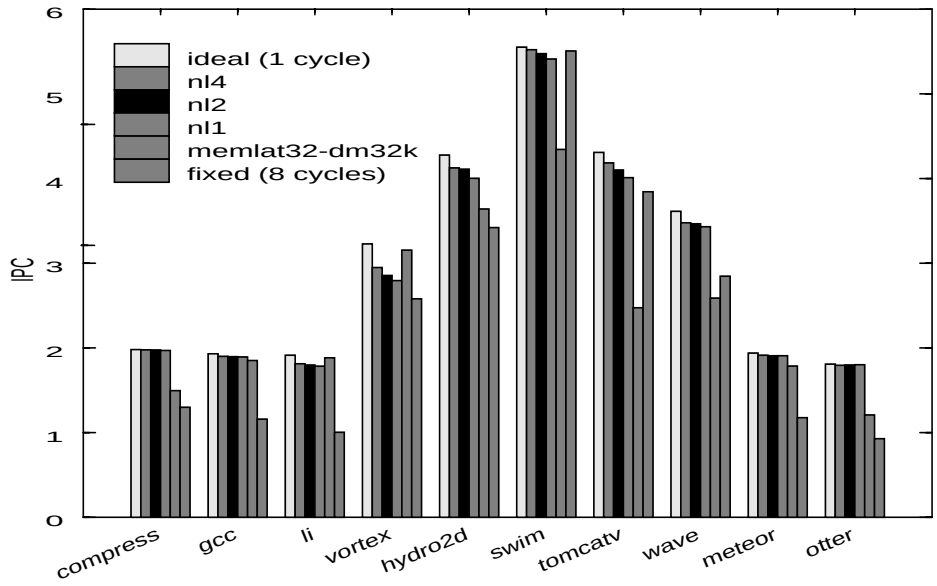


Figure 11: Effects of Number of Loads Completed on IPC

floating point benchmarks and vortex. The other five 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 1% of ideal for fut2 to within 8% of ideal for fut0. Looking at the corresponding latency tolerance graphs in Figure 10, latency tolerance generally increases as we decrease the fill up time. These results match our expectations, 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 a processor with 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 11 shows the impact of these limits on IPC and Figure 12 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 Microarchitecture

To evaluate the impact of various microarchitectural changes on our results, 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 13 shows the effect of various processor configurations on IPC and

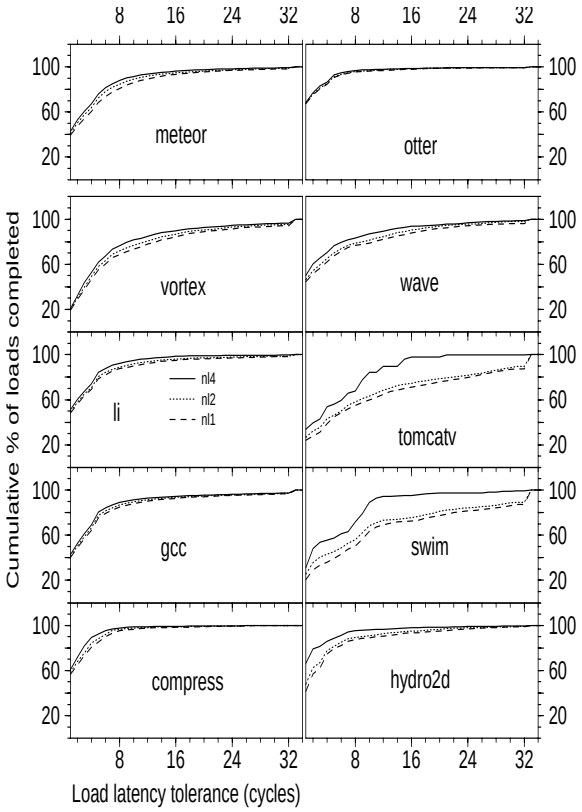


Figure 12: Effects of Number of Loads Completed on Load Latency Tolerance

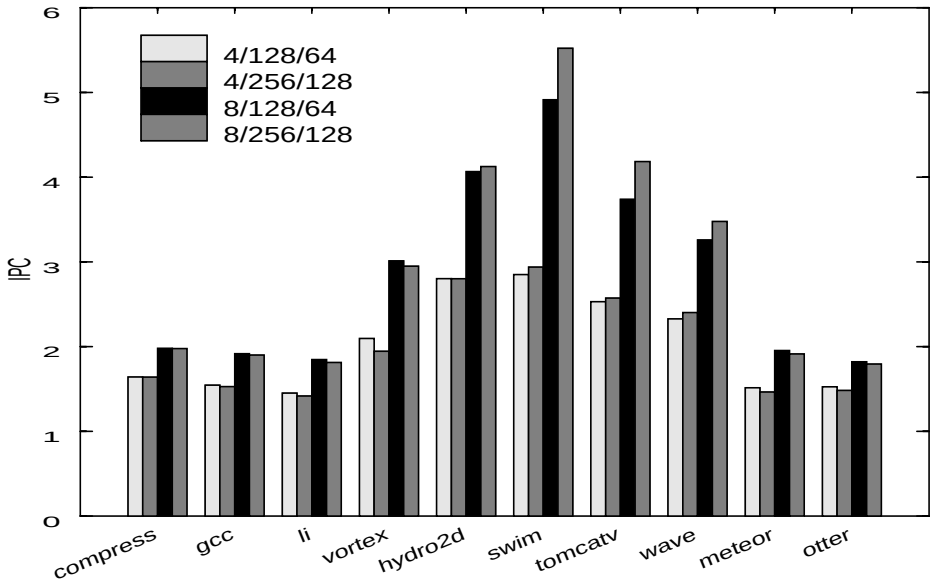


Figure 13: Effects of Processor Microarchitecture (Issue-width/RUU-size/LSQ-size) on IPC

Figure 14 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

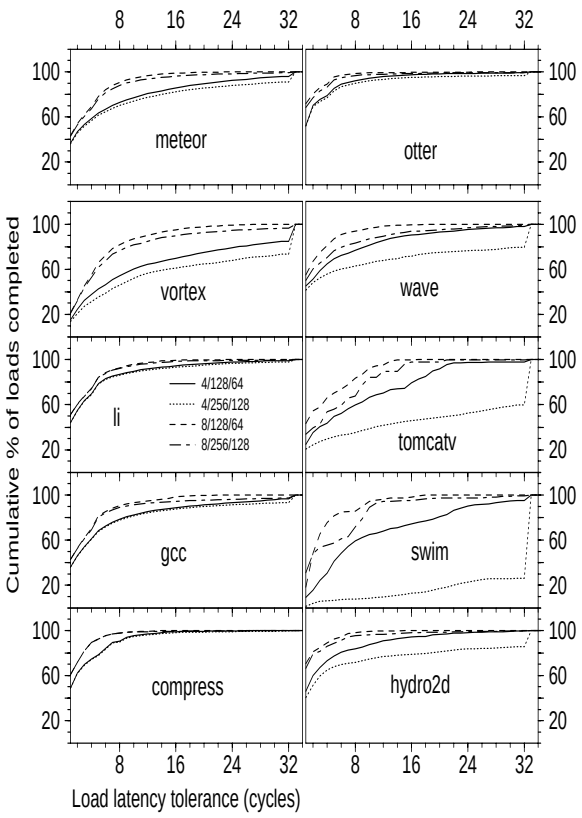


Figure 14: Effects of Processor Microarchitecture on Load Latency Tolerance

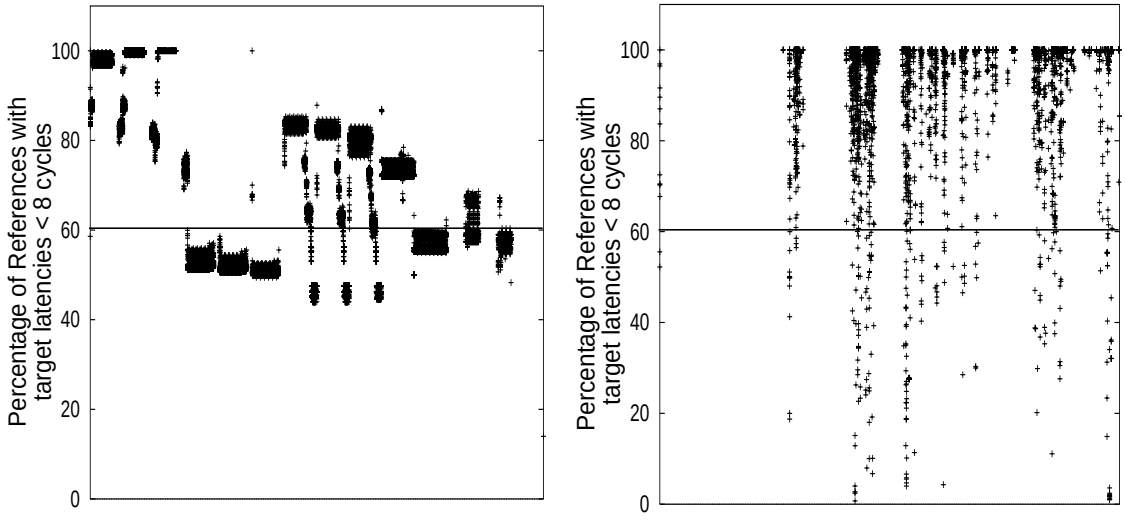
IPC than buffer space does. We note that all the IPC values shown are within 12% of the corresponding processors with an ideal memory system. Also, we see that the IPC values are mostly independent of the number of RUU/LSQ entries. However, the situation is very different with respect to the amount of latency tolerance. From Figure 14, we see that latency tolerance increases when either the issue width decreases or the RUU/LSQ entries increases. The floating point programs exhibit larger increases than the other 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 Load Latency Tolerance Variation

Thus far, we have presented an evaluation of our mechanism for quantifying load latency tolerance. During our latency tolerance computations, the processor determines how long a load can be outstanding without degrading performance. This is the target latency for the load. We gathered a profile of the target latencies of loads over entire runs of programs. Such information can be used to guide processor and/or memory system optimization, study the spatial and temporal locality characteristics of load latency tolerance, etc.

A load can be identified by either:

1. the effective address (EA) that it accesses
2. its program counter (PC)



a) swim - EA distribution

b) gcc - PC distribution

Figure 15: Target Latency Distributions

Profile information based on effective addresses is important because caches are accessed using effective addresses and this information can be used directly for any profile-guided optimizations. On the other hand, many programs consist of loops. The latency tolerance of a load is partly determined by the structure of its dependence graph (particularly branch dependencies), and since dependence graphs can be expected to look alike over different iterations of a loop, PC based statistics tend to repeat periodically. Hence, we obtain a distribution of the target latencies for each effective address as well as each PC for the different benchmarks.

While collecting effective address based profiles, we aggregate groups of adjacent cache blocks into larger blocks called macroblocks, similar to Johnson and Hwu [7], such that the target latencies of cache blocks within each macroblock are similar. We use a macroblock size of 256 bytes. Considering an L2 cache access latency of 8 cycles, it is reasonable to think of all loads with a target latency of less than 8 cycles as being in the critical zone and the rest of the loads as being in the non-critical zone. Figure 15a shows the percentage of references that lie in the critical zone for each macroblock for the benchmark `swim`. Figure 15b shows the same information for the PCs of `gcc`. The benchmarks were chosen because they exhibit the most interesting characteristics in the respective categories.

The horizontal line shown in Figure 15 are set at 60% and can be used to specify the latency tolerance threshold for classifying loads as critical. For example, if at least 60% of references to a PC over the entire run of the program lie in the critical zone, then we could classify the load corresponding to the PC as a critical load. Such a classification can be done using macroblocks as well. Thus, loads above the latency tolerance threshold in the figures are non-critical while those below, are critical. For `swim`, loads to about 40% of all macroblocks referenced, lie in the critical zone. This constitutes about 39% of all references. Similarly for `gcc`, about 24% of all PCs which correspond to 83% of total references are in the critical zone. These results show that there is a lot of variation in the latency tolerance among load instructions.

5.8 Traditional Memory Hierarchies

We now evaluate how good traditional memory systems are in capturing this variation. To determine this, we track which level in a traditional memory hierarchy satisfies each load and increment a counter for the corresponding computed 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 computed latency tolerance less than 8 cycles should be satisfied by the L1 cache. Similarly, loads with computed 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 computed latency tolerance less than 8 cycles. Similarly, the bottom row for each benchmark corresponds to loads with computed 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`, 17% of loads require low latency but are satisfied by the L2 cache, whereas 28% 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 15% to 21% 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` 3% to 34% of loads require low latency yet miss in the L1 cache, whereas 18% to 36% of loads are high latency and hit in the L1. Finally, for the floating point benchmarks a noticeable fraction (1%-6%) of loads 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 3% and 2% low latency L1 load misses in the 32KB cache to 34% and 28%, respectively, in the 8KB cache. Although not shown, reducing the boundary from 8 cycles to 6 cycles can increase the number of low

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

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	13%	13%	13%	63%	66%	69%	21%	18%	15%	0%	0%	0%
	>= 8	0%	0%	0%	2%	2%	2%	1%	1%	0%	0%	0%	0%
gcc	< 8	8%	8%	8%	73%	75%	77%	5%	3%	2%	0%	0%	0%
	>= 8	1%	1%	1%	11%	11%	11%	1%	0%	0%	0%	0%	0%
li	< 8	9%	9%	9%	78%	79%	80%	3%	2%	2%	0%	0%	0%
	>= 8	1%	1%	1%	8%	8%	8%	0%	0%	0%	0%	0%	0%
vortex	< 8	17%	17%	17%	52%	53%	55%	4%	3%	2%	0%	0%	0%
	>= 8	5%	5%	5%	21%	21%	21%	1%	1%	0%	0%	0%	0%
hydro2d	< 8	22%	22%	22%	62%	63%	64%	7%	6%	4%	3%	3%	3%
	>= 8	0%	0%	0%	5%	5%	5%	1%	1%	0%	0%	0%	0%
swim	< 8	5%	5%	5%	21%	38%	52%	34%	**17%**	3%	2%	3%	3%
	>= 8	0%	0%	0%	18%	**28%**	36%	18%	9%	1%	0%	0%	0%
tomcatv	< 8	11%	11%	11%	21%	38%	46%	28%	11%	2%	6%	6%	6%
	>= 8	0%	0%	0%	12%	23%	28%	18%	9%	3%	3%	3%	3%
wave	< 8	26%	26%	26%	37%	44%	49%	18%	11%	6%	1%	1%	1%
	>= 8	3%	3%	3%	7%	12%	13%	8%	3%	1%	0%	0%	0%
meteor	< 8	8%	8%	8%	71$	73%	74%	6%	3%	2%	1%	1%	1%
	>= 8	2%	2%	2%	12%	12%	12%	1%	1%	0%	0%	0%	0%
otter	< 8	4%	4%	4%	70%	73%	78%	21%	18%	14%	1%	1%	1%
	>= 8	1%	1%	1%	2%	2%	3%	1%	1%	0%	0%	0%	0%

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 hierar-

chy meets the application's latency demands. We compute latency tolerance by determining the number of cycles a load could take to complete without adversely affecting performance compared to a processor with an ideal memory system, where all loads complete in one cycle.

Our simulations show that load latency is a function of the number and type of dependent instructions. In particular, mispredicted branches have a significant impact on computed latency tolerance for the integer benchmarks. We also observe that most of the programs we studied do exhibit some latency tolerance, and still obtain IPC values comparable to a processor with an ideal memory system. Our results show that between 1% and 71% of loads must complete in one cycle, and between 7% 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 [14] 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 [1,7,27], 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 particularly, the anonymous referees for their extremely helpful feedback on earlier versions of this paper. This work supported in part by NSF CAREER Award MIP-97-02547, DARPA Grant DABT63-98-1-0001, NSF Grants CDA-97-2637, CDA-95-12356, and EIA-99-72879, 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.

8. References

[1] Santosh G. Abraham, Rabin A. Sugumar, Daniel Windheiser, B. R. Rau, and Rajiv Gupta. Predictability of Load/Store Instruction Latencies. pages 139–152, December 1993.

[2] Todd M. Austin and Gurindar S. Sohi. Dynamic Dependency Analysis of Ordinary Programs. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 342–351, May 1992.

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

[4] George Z. Chrysos and Joel S. Emer. Memory Dependence Prediction using Store Sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 142–153, June 1998.

[5] 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.

[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*, pages 315–326, 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, May 1992.

[10] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value locality and load value prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138 – 147, December 1996.

[11] S. McFarling. "Combining branch predictors". Technical Report WRL Technical Note TN -36, Digital Equipment Corporation, June 1993.

[12] Wen mei W. Hwu and Yale N. Patt. Checkpoint Repair for Out-of-Order Execution Machines. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 18–26, Pittsburgh, Pennsylvania, June 2–5, 1987. IEEE Computer Society TCCA and ACM SIGARCH. *Computer Architecture News*, 15(2), June 1987.

[13] 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.

[14] 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.

[15] A. Nicolau and J. Fisher. Measuring the Parallelism Available for Very Long Instruction Word Architectures. *IEEE Trans. on Computers*, C-33 (11):968–976, November 1984.

[16] Jim Pierce and Trevor Mudge. Wrong-Path Instruction Prefetching. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 165–175, Paris, France, December 2–4, 1996. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[17] 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.

[18] Kevin Skadron and Douglas W. Clark. Design Issues and Tradeoffs for Write Buffers. In *Proceedings of the 3rd International Symposium on High Performance Computer Architecture (HPCA)*, pages 144–155, 1997.

[19] James E. Smith and Andrew R. Pleszkun. Implementation of Precise Interrupts in Pipelined Processors. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 36–44, June 17–19, 1985. IEEE Computer Society TCA and ACM SIGARCH. *Computer Architecture News*, 13(3), June 1985.

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

[21] M. D. Smith, M. Johnson, and M. A. Horowitz. Limits on Multiple Instruction Issue. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 290–302, May 1989.

[22] 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.

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

[24] James E. Thornton. Parallel Operation of the Control Data 6600. In *Proceedings of the Fall Joint Computers Conference*, volume 26, pages 33–40, 1964.

[25] G. S. Tjaden and M. J. Flynn. Detection and Parallel Execution of Parallel Instructions. *IEEE Transactions on Computers*, C-19 (10):889–895, October 1970.

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

[27] 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*, pages 93–103, December 1995.

[28] David W. Wall. Limits of Instructional-Level Parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 176–188, April 1991.

[29] 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, May 1992.