The Effects of Memory-Access Ordering on Multiple-Issue Uniprocessor Performance *

Brian Grayson
bgrayson@ece.utexas.edu

Lizy John
ljohn@ece.utexas.edu

Craig Chase
chase@ece.utexas.edu

Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712-1084

Abstract

We study the effect of memory access ordering policies on processor performance. Relaxed ordering policies increase available instruction-level parallelism, but such policies must be evaluated subject to their effect on memory consistency — since virtually all microprocessors are designed to be compatible with shared memory multiprocessor systems, even uniprocessor desktop computers are constrained by the rules of multiprocessor memory consistency models.

We define the set of potential parallelism-restricting ordering rules found in strong memory models. We then construct a spectrum of possible memory models by progressively relaxing these restrictions. Some of our models are similar to those of existing commercial processors; other models illustrate potential alternatives. We simulate and analyze several uniprocessor benchmarks from the SPEC95 and SPLASH-2 suites using a superscalar processor simulator (Armadillo) developed at the University of Texas at Austin. This simulator models dataflow instruction execution, branch prediction, speculative execution, memory disambiguation and an aggressive memory system.

Our experiments confirm the significant benefits of a weaker memory model on processor performance. Although the absolute performance varies considerably from benchmark to benchmark, the relative performance gains of relaxing specific memory ordering constraints is surprisingly similar across most of the benchmarks.

* Brian Grayson was supported in part by a National Science Foundation Graduate Fellowship. This research was supported in part by the National Science Foundation under grants CCR-9796098 and EIA-9807112, and by an equipment grant from Intel.

1 Introduction

In recent years, the gap between memory performance and CPU performance has widened considerably, and is expected to continue to do so. Several "transparent" remedies to hide the main memory latency and improve bandwidth have become quite common or even indispensable, such as caching, wide buses, wide memories, and interleaved memories. Further improvements in the memory system are possible, but most are not transparent, and thus require more careful consideration from both system designers and the programmers on those systems.

One such improvement is allowing memory references to be initiated out of order by the processor. However, such a reordering may change the processor's memory consistency model, and requires nontrivial processor overhead to avoid data hazards such as "true data" dependences (RAW) and "anti" dependences (WAR). While such a weakening provides improved performance for uniprocessor applications, this must be balanced against the cost of implementation. In addition, since most recent superscalar processors are designed to be used in shared memory multiprocessors, a weaker memory consistency model will alter the shared-memory interactions between the processors, and may be incompatible with software that was written assuming a stronger consistency model.

This paper quantifies the performance improvement of various degrees of aggressive memory-reference reordering policies. In specific, the processor model (in addition to being superscalar with out-of-order issue and completion) will contain multiple load/store units capable of out-of-order issue and completion of memory references, and the non-blocking caches will be capable of handling dozens of pending memory references and multiple concurrent split transactions. Such a model should help expose fundamental characteristics of memory access reordering.

Note that while much previous research has examined the performance tradeoffs of various memory consistency models, most of the research (for example, [9, 2, 19]) has centered on shared-memory machines. The impact of various consistency models, including the penalties of memory disambiguation, on *uniprocessor* performance has not been adequately studied.

Other methods of reducing the effect of the CPU-memory performance gap have been suggested. Previous research has examined the impact of allowing loads to be performed ahead of earlier accesses, through techniques such as a write buffer [20, 13] and nonblocking caches [5]. McKee, Wulf *et al.* [17, 18] examined memory access ordering in the area of multiple stream buffers in a vector computer. The decoupled access/execute [21, 12] architectures process memory accesses and scalar computations in different units to allow explicit overlap of computation and memory access overhead. Lipasti *et al.* [16] proposed exploiting "value locality" for loads, that is, predicting the loaded value based on past behavior.

2 Background

A memory consistency model limits what memory access reorderings are allowed by the processor. For example, Sequential Consistency requires that all memory accesses appear to be performed in strict program order. Although the term "memory consistency model" is typically restricted to discussions of shared-memory multiprocessors, that term will be used throughout this paper to signify the memory access reordering policy.

We consider processors with a traditional sequential instruction stream (*i.e.* non-VLIW). Instruction-level parallelism is detected by the processor as instructions are decoded. Two instructions can be executed in parallel or out of order as long as they do not have a data dependence. Our simulations assume an infinite number of registers available for renaming, thus the only limitations to parallelism are:

- true data dependences between instructions,
- branch prediction accuracy and the number of conditional branches the model will speculate down,
- the number of functional units available, and
- memory ordering constraints imposed by memory hazards and the memory consistency model.

2.1 Memory Hazards and Memory Disambiguation

In our study we consider several variations on the memory ordering constraints along with four possible machine configurations (*i.e.* number of functional units, latency to the caches) and measure the performance of several SPEC95 [1] and SPLASH-2 [22] benchmarks. Note that, regardless how weak the memory model, we must restrict the order of load and store instructions to avoid memory hazards. A read-after-write (RAW) hazard occurs when the instruction stream contains a store to an address followed by a load to the same address. A write-after-read (WAR) hazard occurs when the load is issued before the store, and a write-after-write (WAW) hazard occurs when two stores are issued to the same address. In this study we do not consider the possibility of renaming memory locations, and thus always prevent loads and stores from being reordered when a memory hazard would be produced.

A major impact of memory hazards is the memory disambiguation problem, which can severely limit instruction-level parallelism in some applications. For example, a store instruction cannot safely be issued unless the address is known for all previously decoded load and store instructions. Until all addresses are known, the possibility of a WAR or WAW hazard cannot be eliminated. Similarly, a load instruction cannot be safely issued unless the address is known for all previously decoded store instructions. Our study accurately considers the effect of memory disambiguation. The simulator is a functional simulator, and thus models the execution of every instruction, including the instructions involved in the calculation of effective addresses for memory operations. Trace-driven simulations generally are incapable of accounting for memory disambiguation effects and thus can report higher degrees of parallelism than would actually be obtained in a more detailed simulation.

2.2 Limits to Instruction Reordering Imposed by the Consistency Model

In addition to limiting loads and stores *to the same address*, we study the effect of memory models that may restrict the reordering of loads and stores to different addresses. Given two memory reference instructions decoded in some sequence, we can identify four possibilities: a load followed by another load, a store followed by a load, a load followed by a store, and a store followed by another store. For each of these four cases, we consider the possibility of three types of ordering restriction, summarized in Table 1. The A (for "always") restriction type requires that the instructions must be executed in sequence, regardless of their effective addresses. The E (for "effective address") restriction type permits the two instructions to be issued in parallel or out of order as long as they are to different effective addresses. The N (for "never") restriction indicates that the execution of the two instructions is never constrained, and can occur in parallel or in any order.

A	Always delay the execution of the later instruction until the earlier instruction completes.
E	Only delay the execution of the later instruction until the earlier instruction completes if its effective address matches the effective address of the earlier access.
N	The later instruction does not need to wait for the completion of the earlier instruction, even if either effective address is currently unknown.

Table 1: Classification scheme for memory reference reordering dependences

Given four possible instruction sequence pairings (*e.g.* a load followed by store), and three ordering restriction types (A, E or N), we have, at first glance, a total of 3^4 or 81 possible memory models. We designate an individual model with a four-character sequence. Each character is one of A, E or N and designates the ordering restriction type for one of the four sequence pairings as described below:

1st character position:	load followed by another load
2nd character position:	store followed by a load
3rd character position:	load followed by a store
4th character position:	store followed by another store

For example, the AAAA model, which is the most restrictive, is strong enough to guarantee sequential consistency. By contrast the NEEE model, the weakest model that we consider, cannot implement any memory model stronger than weak consistency. Since we do not permit memory hazards, the N-type of memory ordering restriction is not permitted in any of the last three character positions. Thus, the number of memory models is actually $3 * 2^3$ or 24. Of these hypothetical models, several do not correspond to reasonable configurations. For example, the AEEE model requires that loads always be performed in order with respect to other loads, but permits any other type of reordering. There are no practical multiprocessor memory consistency models, or hardware architectures, for which this combination of restrictions makes sense.

Within our framework, it is easy to determine which memory models are "stronger than" other memory models and which are incomparable. Note that the stronger-than relation defines a partial order (not a total order) over the memory models. For example, while both AEAA and EAAA are weaker than AAAA, no comparison can be made between the strength of AEAA and EAAA. As stated above, AAAA is the strongest possible model under our naming scheme. We refer to this model as the sequential consistency model, although technically speaking, AAAA is a memory order restriction policy that is sufficient for sequential consistency (it may not be a necessary condition for all architectures or applications). Similarly, we refer to the weakest possible model, NEEE, as the weak consistency model.

The set of ten models under investigation are described below. Figure 1 shows the partial order for these models. For clarity, any "stronger than" relation implied by transitivity has been eliminated from Figure 1. Note that the sequence of "weakenings" to progress from one model to another is not unique. Six of the models are given "natural" names, to aid the reader in mapping our results onto traditional memory consistency models.

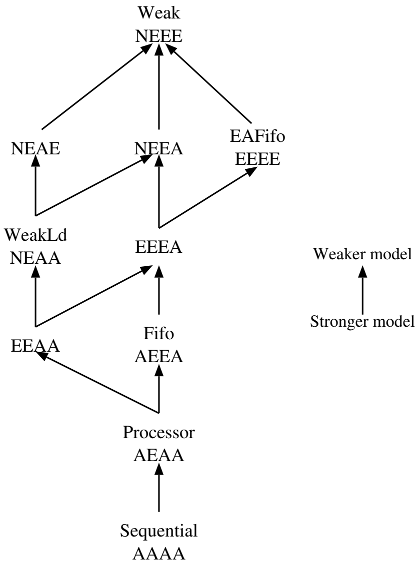

Figure 1: Partial order of the modeled consistency models

AAAA, or Sequential Consistency: all accesses are performed strictly in decode order (sufficient conditions for sequential consistency [14]).

AEAA, or Processor Consistency: all stores are performed in order with respect to stores, and all loads are performed in order with respect to loads. However, if the oldest load and an earlier store are for different addresses, the load can be performed early (sufficient conditions for processor consistency [6]).

EEAA: all stores are performed in order, but loads can be reordered with respect to both stores and loads to different addresses.

AEEA, or Fifo: all stores are performed in order, and all loads are performed in order. However, if the oldest load and the oldest store are for different addresses, they can be performed out-of-order with respect to each other.

EEEA: if a load and any other access are for different addresses, they can be performed out-of-order with respect to each other. Stores are performed in order with respect to other stores.

NEEA: loads can always be reordered with respect to other loads, even if one or both effective addresses are not known or if both addresses are the same. Loads can also be reordered with respect to earlier stores if the effective addresses are different. Stores are performed in order with respect to other stores.

EEEE, or EAFifo: Any two memory accesses for different addresses can be reordered. In other words, all memory accesses to the same effective address must be performed in order, even if both accesses are loads.

NEAA, or WeakLoad: a weakening of Processor Consistency, where loads are not required to be ordered with respect to each other. In other words, WeakLoad (NEAA) has the load behavior of weak consistency and the store behavior of processor consistency.

NEAE: a weakening of WeakLoad (NEAA) such that stores to different addresses can be reordered with respect to each other. (Equivalently, weak consistency with the additional rule that stores can not be performed before preceding loads, even if the effective addresses are different.)

NEEE, or Weak Consistency [4]: accesses can be performed in any order provided no memory hazards are introduced.

Several commercial microprocessors can be approximately classified according to this classification scheme. For example, the HP PA-RISC 1.1 architecture specifies sequential consistency, or AAAA. The Pentium Pro is roughly equivalent to our NEAA model — stores are still performed in order with respect to other accesses. The PowerPC and the MIPS IV guarantee nothing stronger than weak (NEEE) consistency. Note, however, that some of these processors may have ordering policies besides those in the classification. For example, the Pentium Pro allows loads to be speculatively performed early in some situations that may be unsafe, and simply squashes the load if it later finds out that the load would have violated the memory consistency model [11]. Thus, the Pentium Pro provides the appearance of Processor Consistency (AEAA) while obtaining performance close to that of NEAA.

3 Simulation Methodology

3.1 Simulator

The Armadillo simulator [7] was used for this research. The simulator is execution-driven, that is, it reads an executable and simulates the execution of every instruction. A trace-driven simulator (where the trace consisted simply of a list of effective addresses) would not provide as accurate results. Most traces typically do not contain sufficient information for the simulator to calculate the impact of stalls due to memory disambiguation (unknown-effective-address) dependences, and effects caused by (mispredicted) speculative execution.

3.2 Processor Model

The processor model is a superscalar processor that uses the PowerPC instruction set and supports register renaming, speculative execution, out-of-order execution and true out-of-order retirement/graduation/commitment[1]. The branch prediction algorithm is two-level adaptive [23] with a single global branch history register. We modeled two CPU configurations, the "simple" and "advanced" models, whose characteristics are presented in Table 2. The parameters for the "simple" model were chosen to be roughly comparable to currently-available processors, while the "advanced" model has a slightly more complex CPU. We intentionally did not vary many characteristics of the memory system (cache sizes and associativity), to limit the number of independent variables under investigation.

	Simple model	Advanced model
# active insns	32	128
#IU/#FPU	2/2	4/4
#LSU/#L1 ports	2/2	4/4
spec. depth	3 branches	5 branches
BHT entries	1024	16384
bits of history	4	6
L1 cache	0- or 3-cycle 16KB 4-way, with 32-byte line size	
L2 cache	6-cycle (3-1-1-1) 512KB 4-way, with 32-byte line size	
Memory	7-1-1-1 timing on 3:1 64-bit bus, or 21-3-3-3	

Table 2: Configuration characteristics

All levels of the memory hierarchy support the features of an aggressive memory-reference reordering system: full

[1]Most commodity processors perform in-order retirement, in order to support precise interrupts.

memory address disambiguation, nonblocking caches, support for concurrent accesses, and split-transaction buses.

We modeled L1 hit times of 0 cycles and 3 cycles, to examine the ability of a weaker memory consistency model to hide latency. Besides providing a more powerful CPU, the advanced configuration provides four ports to the L1 cache versus two ports for the simple configuration. All L1 ports can handle either a load or a store in every cycle.

3.3 Benchmarks

We utilized several benchmarks from the SPEC95 [1] and SPLASH-2 [22] suites. Note that in this study, we configured the SPLASH-2 benchmarks for uniprocessor execution. All benchmarks were compiled using `gcc-2.7.2.1 -O3` and `binutils-2.7`, both configured for the `powerpc-unknown-elf` target. The SPECfp benchmarks were first translated to C using `f2c -a`, and a few highly-used utility routines in the fmm and fpppp benchmarks were modified to eliminate redundant memory accesses that `gcc` was not optimizing away.

Rather than truncating the simulations after the first 100 million non-initialization instructions, or sampling 100 million from the first 1 billion such instructions (both of which can produce inaccuracies [3]), we chose to reduce the problem sizes so that the benchmark performed around 50 to 100 million instructions between initialization and cleanup/final output.

The running time and the number of loads and stores performed by each benchmark are reported in Table 3. The first three benchmarks are from SPECint95, the next four are from SPECfp95, and the remaining four are from SPLASH-2 (run on a single processor).

Program	Parameters	# of insns	# of loads	# of stores
go	"10 9 null.in"	136M	28.9M	12M
compress	30KB, 25 iters	105M	21M	17M
li	6-queens, 2 iters	112M	27M	21M
tomcatv	N=90, 60 iters	139M	34M	8.4M
swim	256x256 grid	91M	36M	8.4M
mgrid	32x32x32, 7 iters	96M	31M	2.0M
fpppp	3 atoms	49M	13M	4.7M
barnes	256 bodies	91M	4.2M	3.2M
fmm	768 particles	66M	16M	14M
ocean	66x66, 5 iters	109M	24M	7.8M
water-nsq	27 mol's, 3 iters	64M	3.1M	1.6M

Table 3: Benchmark Summary

3.4 Caveats

For this paper, we assume a 0-cycle perfect and prefetching instruction cache.

Also, the simulator supports a restricted form of load forwarding from the write buffer. Note that allowing a load to be satisfied from the write buffer slightly weakens a literal interpretation of our memory ordering policies, but load forwarding is a common optimization of commercial superscalar processors.

In our model, an access must wait for all dependent accesses to complete before it can be issued to the cache. Several commercial processors allow the second access to be pipelined to the cache, and simply stall the cache's pipeline as needed to enforce the memory consistency model. This effect may add a few cycles of latency under some situations in our simulations.

4 Experimental Results and Analysis

For each of the four processor configurations (simple or advanced CPU, each with a 0- or 3-cycle L1), simulations were performed for the ten different memory consistency models for the eleven benchmarks. To summarize this large amount of simulation data, we took the harmonic mean across the SPECint95 and SPECfp95/SPLASH-2 benchmarks of the achieved IPC (instructions per cycle) for each memory consistency model and configuration. The results for these simulations are shown in Figure 2 for SPECint95 and Figure 3 for SPECfp95 and SPLASH-2.

Each graph shows the attained IPC for the memory models by plotting a point at the appropriate location in the y-direction. Notice that the x-axis does not provide information — the horizontal spacings simply place the nodes in correlation with the partial order of the memory models, as shown in Figure 1. By examining how each performance graph is distorted compared to the original partial order in Figure 1, one is able to easily visualize which single-step relaxations of the memory model provide the largest jump in performance.

The graphs for the four different configurations (simple or advanced, 0-cycle or 3-cycle L1) are shown together to allow easy comparisons. The placement of the graphs is described in Table 4. Specifically, in each group of four graphs, the 3-cycle L1 results are in the left graphs, and the simple CPU results are in the upper graphs. Note that the y-axis scales and offsets for the graphs on a particular page are the same, so that comparisons can be easily performed.

As an example of how to interpret the graphs, examine the graphs for the SPECint harmonic means, in Figure 2. By comparing the upper graphs with the respective lower graphs, we see that going from the simple CPU configuration to the more advanced CPU configuration produces a

Simple CPU, 3-cycle L1	Simple CPU, 0-cycle L1
Advanced CPU, 3-cycle L1	Advanced CPU, 0-cycle L1

Table 4: Configurations for the IPC graphs

small performance improvement, and does not modify the shape of the graph significantly. On the other hand, comparing the left graphs with the right graphs, reducing the L1 latency from 3 cycles to 0 cycles provides relatively uniform IPC improvements, regardless of the memory consistency model or the CPU configuration — the graph shifts up around .5 IPC. These are both consistent with the expected behavior of integer benchmarks — first, the code is more likely to have its performance determined by the critical path of instructions, so adding additional functional units (from upper graphs to lower graphs) does not significantly reduce the execution time; and second, reducing the latency of the L1 (from left to right graphs) allows all of the loads and stores on the critical path to be completed sooner, leading to an overall decrease in execution time.

Space limitations prevented us from including the graphs for all of the benchmarks, so we will simply present an overall analysis, along with a discussion of one case of unexpected behavior. An extended version of this paper [8] contains the full set of graphs for each benchmark, and a more detailed analysis.

4.1 General Trends

Observations from the harmonic-mean graphs include:

- The SPECint95 benchmarks have significantly different behavior than the floating-point benchmarks (as expected). Specifically, as mentioned above, the integer benchmarks are more likely to have performance limited by critical paths through the code, so reducing the L1 latency provides the highest performance improvement. For the floating-point benchmarks, the code is more likely to contain regular loops, where the degree of parallelism is much higher. The 3-cycle L1 latency does not affect performance as much because the latency can be hidden through the pipelining of independent memory accesses and execution of independent instructions (dynamic loop unrolling/pipelining).

- Similarly, for the floating-point benchmarks, the more advanced CPU configuration provides a significant performance improvement (around .3 IPC) over the simple CPU configuration for the weaker memory models — the floating-point benchmarks are resource-constrained, so each additional CPU resource allows more instruction-level parallelism (ILP).

- For the floating-point benchmarks, weakening the memory model provides a very large performance improvement. For each of the configurations, the performance gap between either sequential or processor consistency and the weakest consistency models is around a factor of two, or an improvement of around .5 to 1.0 IPC. For the integer benchmarks, the gap is smaller, but still around .3 to .4 IPC.

- For both sets of benchmarks, two single-step relaxations of the memory consistency model provide the highest incremental performance: the Axxx to Exxx transition (allowing loads to bypass older loads), and the xxxA to xxxE transition (allowing stores to bypass older stores).

- For both sets of benchmarks, there is little performance difference between the NExA and EExA models (these appear around the middle of the IPC graphs). This hints that both the xxAx to xxEx transition (where stores can bypass loads) and the Exxx to Nxxx relaxation (where loads can be performed ahead of previous loads whose effective addresses are not yet known) do not have a major effect on performance. (The latter relaxation does have a significant impact on performance in mgrid and tomcatv, as discussed below, and on the FP harmonic mean in the advanced 0-cycle L1 graph.)

- Also, qualitatively, the graphs become taller when moving from a fast L1 to a slow L1 (right to left) or when improving the CPU from the simple to the advanced configuration. Specifically, as the L1 is slower, the weaker memory models are able to widen the performance gap between themselves and the stronger memory models – they are able to successfully hide more of the L1 latency than the stronger models. Similarly, as the CPU configuration becomes more advanced, the processor is capable of exposing a larger instruction window, so the weaker memory models are able to find and exploit more memory-access parallelism.

  Another view of this is that, as the L1 becomes faster or the CPU becomes less powerful, the performance gap between the weakest and strongest memory models lessens because there is not as much memory-access parallelism to exploit — with a faster cache or a less-advanced processor, the bottleneck partially shifts from the memory accesses to the arithmetic functional units (or other CPU-resource constraints).

4.2 tomcatv anomaly: EAFifo can outperform Weak

For tomcatv under the simple CPU configuration, the EEEE to NEEE relaxation actually hurts performance by around .2 IPC. This is due to a "priority inversion" effect — more parallelism allows a reordering such that a critical-path operation incurs more delay due to a reordered non-critical operation.

A simple example of priority inversion is due to L1 conflict misses. Consider a direct-mapped cache, and the following code, where initially the data for 0x1000 is in the cache, addresses 0x1000 and 0x2000 map to the same line, and register 1 holds the value 0:

```
add  register1 and 0x1000, and put
       result in register1
load from address in register1
load from 0x2000
```

The effective address calculation of register1 will delay the first load from being performed for a cycle, while the second load could be performed immediately. Under EEEE, the second load must wait until the first load's effective address is known, at which point the two accesses are attempted in parallel (provided both L1 ports are available). The first load hits in the cache, and the second load misses. The overall cost of this code is 1 cycle plus a cache hit and a cache miss. However, under NEEE, the second load is performed early, and causes 0x1000 to be evicted from the cache. When the first load is performed, it again misses, and evicts 0x2000 from the cache. The overall cost is thus 2 cache misses, or an additional penalty of nearly one full cache miss for the weaker memory model.

A 4-way cache helps alleviate this problem. However, the innermost loops in tomcatv reference several arrays in such a way that six different cache lines all hash to the same set in a 16Kbyte 4-way cache. tomcatv, as compiled via `f2c` and `gcc`, is thus susceptible to this priority-inversion effect. This effect was noted by Lebeck and Wood in [15]. Re-executing the simulation with an appropriately-padded array eliminated this anomaly, such that the Weak memory consistency model again outperformed all others.

5 Conclusion

Our experiments confirm that memory access ordering plays a major role in superscalar processor performance. Using the results from a detailed functional simulator, we have presented a comprehensive assessment of the performance penalties of memory access policies. We consider ten distinct policies (*i.e.* memory models) which we feel cover the spectrum of what is reasonable for superscalar processors. We simulate four processor configurations: from modestly parallel with a slow cache, to aggressively parallel with a fast cache. The performance trends revealed in our study extrapolate to much of the realizable design space for current and next generation superscalar architectures.

Our study accounts for dynamic (hardware) memory disambiguation costs, and models execution time penalties caused by speculative execution down incorrectly predicted branch paths. No previously published studies on memory access ordering effects have contained results with this detail or accuracy. Our experiments use applications taken from the SPEC95 and SPLASH-2 benchmark suites.

Not surprisingly, the integer applications perform significantly differently than the floating point applications. Integer benchmarks tend to have much less instruction-level parallelism available. Nevertheless, our study clearly shows that weakening the memory model provides significant performance gains for integer applications by increasing their ability to tolerate memory latency. The harmonic mean taken from our integer benchmarks shows that the big gains in performance come at specific access relaxations: allowing loads to bypass stores, allowing loads to bypass other loads, and allowing stores to bypass other stores. Surprisingly, the gain provided by reordering stores was especially significant. This jump is in part attributable to our literal interpretation of "performing stores in order." Until a store is committed to the cache, all subsequent store operations remain pending, consuming resources and limiting parallelism. Architectures with sophisticated write buffers can achieve relatively strong memory models (*e.g.* Pentium Pro) and still "reorder" stores by committing data into the write buffer early. On those architectures, the benefits of reordering stores may be less pronounced.

Floating point applications also show a strong correlation between high performance and the relative weakness of the memory model. Whereas floating point benchmarks showed significantly different sensitivities with respect to the processor configuration (number of functional units, latency of the cache), the relative performance of the different memory models is strikingly similar to the integer benchmarks. This result is surprising, since the increased performance of the floating point benchmarks comes less from latency tolerance as from the ability of the processor to make use of increased number of memory ports as the memory model weakens. The similarity between the shape of our memory access performance plots for integer and floating point benchmarks is good news to designers, who can adopt the same "rules of thumb" regarding memory access policies for both integer and floating point calculations.

References

[1] Information on the SPEC95 suite of benchmarks is available from `http://www.specbench.org/osg/cpu95/`.

[2] Sarita Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. Technical Report WRL Research Report 95/7, Digital Western Research Laboratory, 1995.

[3] Bryan Black, Andrew S. Huang, Mikko H. Lipasti, and John Paul Shen. "Can Trace-Driven Simulators Accurately Predict Superscalar Performance?" In *1995 IEEE International Conference on Computer Design (ICCD 95)*, 1995.

[4] Michel Dubois, Christoph Scheurich, and Fayé Briggs. "Memory Access Buffering in Multiprocessors," In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, 1986.

[5] Keith I. Farkas, Norman P. Jouppi, and Paul Chow. "How Useful are Non-blocking Loads, Stream Buffers and Speculative Execution in Multiple Issue Processors?" In *First IEEE Symposium on High-Performance Computer Architecture (HPCA 1995)*, 1995.

[6] James R. Goodman. Cache Consistency and Sequential Consistency. Technical Report no. 61, SCI Committee, 1989 (also available as University of Wisconsin-Madison Technical Report CS-TR-91-1006).

[7] B. Grayson. Armadillo: A High-Performance Processor Simulator. Master's thesis, The University of Texas at Austin, 1996.

[8] Brian Grayson, Lizy John, and Craig Chase. The Effects of Memory-Access Ordering on Multiple-Issue Uniprocessor Performance. Technical Report TR-PDS-1998-007, Department of Electrical and Computer Engineering, The University of Texas at Austin, July 1998. Available from `http://maple.ece.utexas.edu/-TechReports/1998/TR-PDS-1998-007.ps.Z`.

[9] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. "Comparative Evaluation of Latency Reducing and Tolerating Techniques," In *18th International Symposium on Computer Architecture (ISCA 1991)*, pages 250–263, 1991.

[10] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, 1990.

[11] Intel Corporation. *A Tour of the Pentium Pro Processor Microarchitecture*. Available on-line at `http://www.intel.com/procs/p6/p6white/-p6white.htm`.

[12] Lizy Kurian John, Vinod Reddy, Paul T. Hulina, and Lee D. Coraor. "Program Balance and its Impact on High Performance RISC Architectures," In *First Symposium on High Performance Computer Architecture (HPCA-1)*, January 1995.

[13] Mike Johnson. *Superscalar Microprocessor Design*. P T R Prentice Hall, 1991.

[14] Leslie Lamport. "How to make a multiprocessor computer that correctly executes multiprocess programs," In *IEEE Transactions on Computers*, volume C-28(9), pages 241–248, 1979.

[15] Alvin R. Lebeck and David A. Wood. "Cache Profiling and the SPEC Benchmarks: A Case Study," *IEEE Computer*, Vol. 27, No. 10, 15–26, October 1994.

[16] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. "Value Locality and Load Value Prediction," In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, October 1996.

[17] Sally A. McKee, Robert H. Klenke, Andrew J. Schwab, Wm. A. Wulf, Steven A. Moyer, James H. Aylor, and Charles Y. Hitchcock. "Experimental Implementation of Dynamic Access Ordering," In *Proceedings of the 27th Hawaii International Conference on Systems Sciences (HICSS-27)*, January 1994.

[18] Sally A. McKee and Wm. A. Wulf. "Access Order and Memory-Conscious Cache Utilization," In *First Symposium on High Performance Computer Architecture (HPCA-1)*, January 1995.

[19] Vijay S. Pai, Parthasarathy Ranganathan, Sarita V. Adve, and Tracy Harton. "An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors," In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 12–23, 1996.

[20] Kevin Skadron and Douglas W. Clark. "Design Issues and Tradeoffs for Write Buffers," In *Third Symposium on High Performance Computer Architecture (HPCA-3)*, February 1997.

[21] J. E. Smith, S. Weiss, and N. Y. Pang. "A Simulation Study of Decoupled Architecture Computers," In *IEEE Transactions on Computers*, August 1986.

[22] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. "The SPLASH-2 Programs: Characterization and Methodological Considerations," In *Twenty-second International Symposium on Computer Architecture (ISCA 1995)*, pages 24–36, June 1995.

[23] Tse-Yu Yeh and Yale N. Patt. "A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History," In *20th International Symposium on Computer Architecture (ISCA 1993)*, pages 257–266, San Diego, CA, 1993.

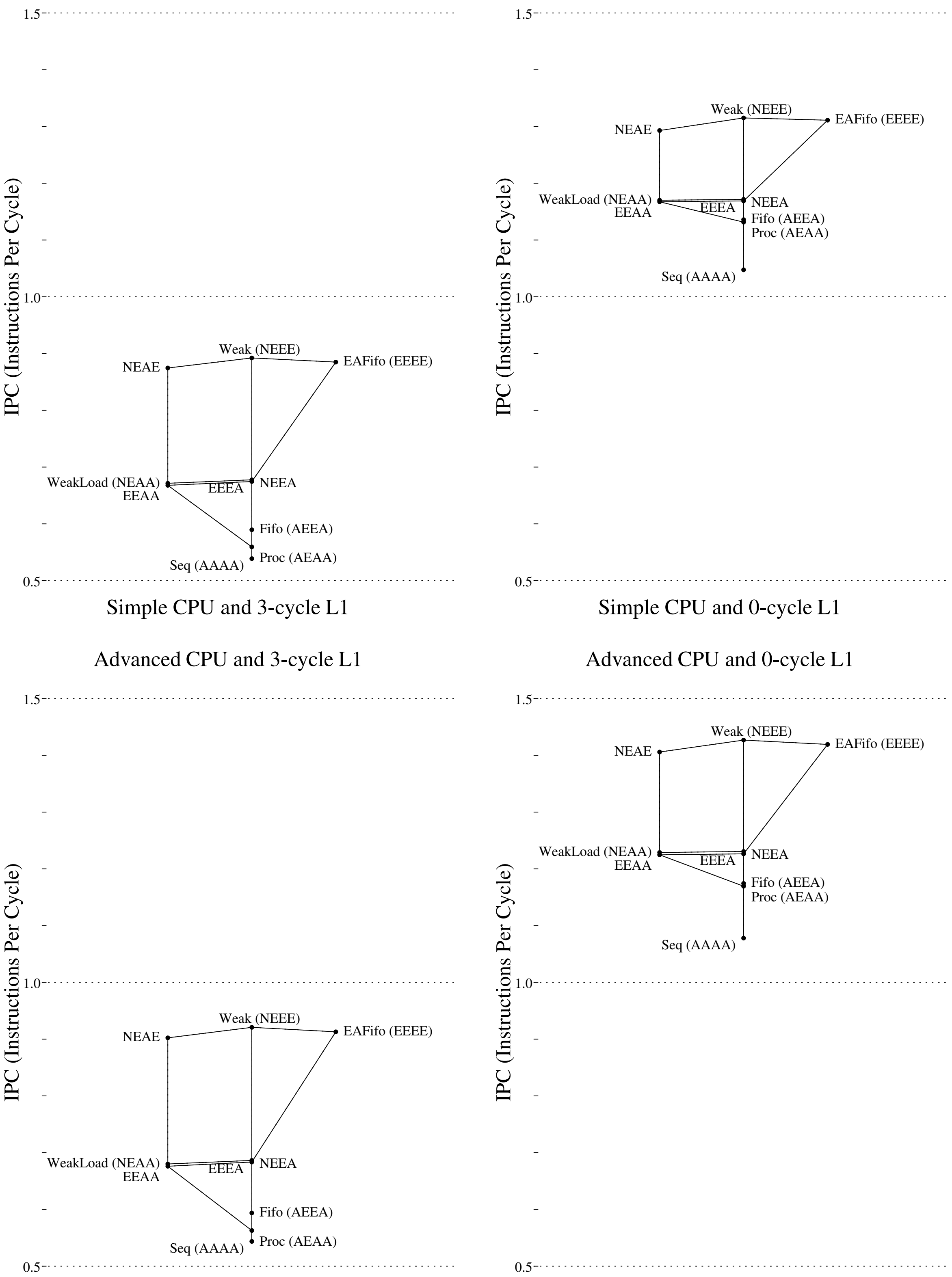

Figure 2: IPC results for SPECint95 (harmonic mean)

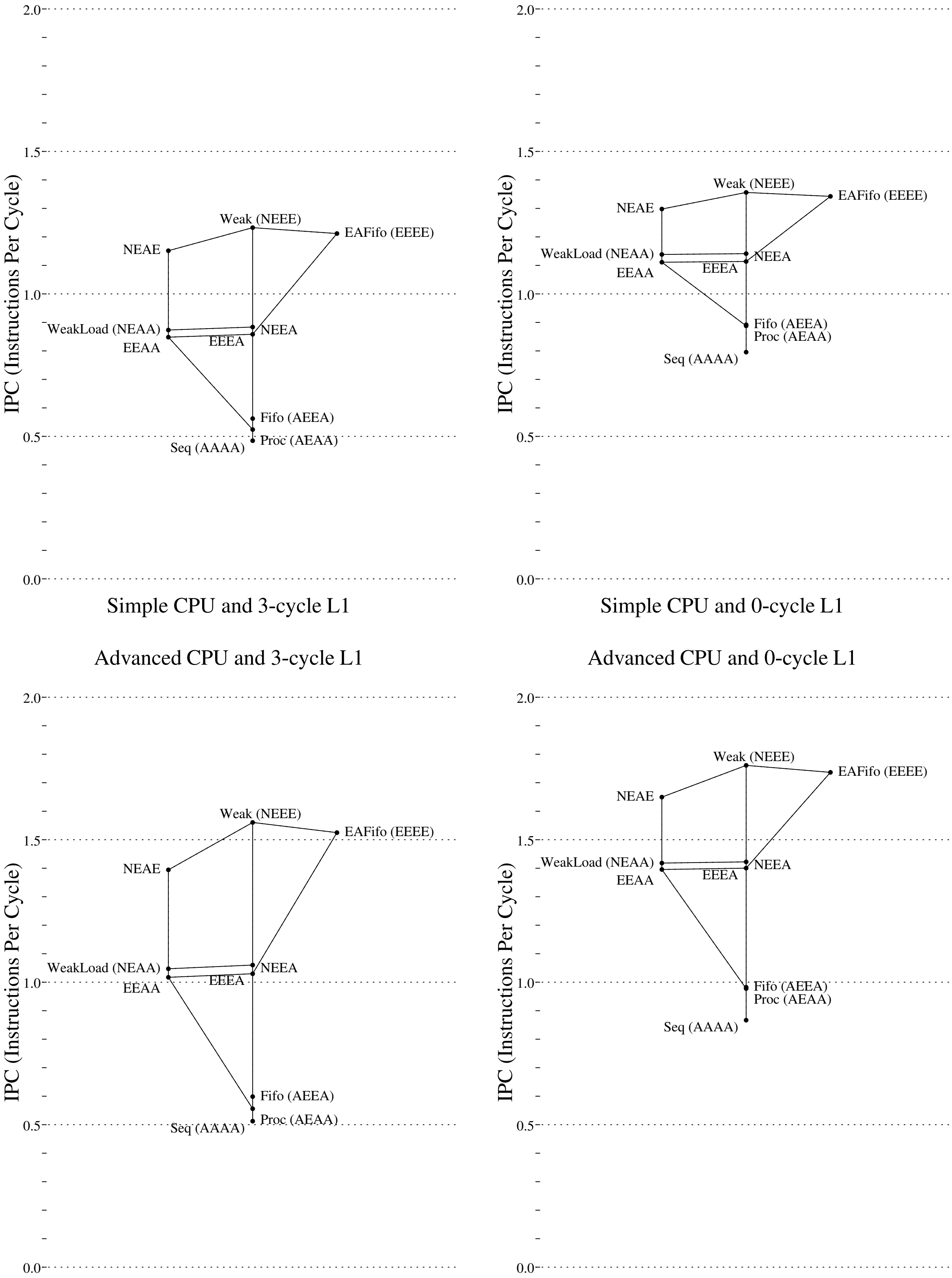

Figure 3: IPC results for SPECfp95 and SPLASH-2 (harmonic mean)