RICE UNIVERSITY

An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors

by

Parthasarathy Ranganathan

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Approved, Thesis Committee:

Sarita V. Adve, Chair
Assistant Professor in Electrical and
Computer Engineering

Keith Cooper
Associate Professor of Computer Science

Willy Zwaenepoel
Professor of Computer Science

Houston, Texas

April, 1997

An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors

Parthasarathy Ranganathan

Abstract

The memory consistency model of a shared-memory multiprocessor determines the extent to which memory operations may be overlapped or reordered for better performance. Studies on previous-generation shared-memory multiprocessors have shown that relaxed memory consistency models like release consistency (RC) can significantly outperform the conceptually simpler model of sequential consistency (SC). Current and next-generation multiprocessors use commodity microprocessors that aggressively exploit instruction-level parallelism (ILP) using methods such as multiple issue, dynamic scheduling, and non-blocking reads. For such processors, researchers have conjectured that two techniques, hardware-controlled non-binding prefetching and speculative reads, have the potential to equalize the hardware performance of memory consistency models. These techniques have recently begun to appear in commercial microprocessors, and re-open the question of whether the performance benefits of release consistency justify its added programming complexity.

This thesis performs the first detailed quantitative comparison of several implementations of sequential consistency and release consistency optimized for aggressive ILP processors. Our results indicate that although hardware prefetching and speculative reads dramatically improve the performance of sequential consistency, the simplest RC version continues to significantly outperform the most optimized SC version. Additionally, the performance of SC is highly sensitive to the cache write policy and the aggressiveness of the cache-coherence protocol, while the performance of RC is generally stable across all implementations. Overall our results show that RC hardware has significant performance benefits over SC hardware, and at the same time, requires less system complexity with ILP processors. Memory write latencies that hardware prefetching and speculative loads are unsuccessful in hiding are the main reason for the performance difference between SC and RC.

Acknowledgments

I would like to thank my advisor, Sarita Adve, for her invaluable guidance and mentoring throughout this work. Her encouragement and enthusiasm were primarily responsible in making this work possible. I would also like to thank my other committee members, Willy Zwaenepoel and Keith Cooper, for their encouragement and suggestions. A significant part of this thesis evolved from my joint work with Vijay Pai. Many thanks are due to him for the many enriching and interesting conversations during the course of our research. I would also like to thank all my friends at Rice who made working at school an enjoyable experience. Finally, I would like to thank my parents and my elder brother for all their support and encouragement which enables me to be what I am today.

Contents

Abstract ii

Acknowledgments iii

List of Illustrations vi

1 Introduction 1

1.1 Motivation 1

1.2 Contributions of this Thesis 3

1.3 Organization 4

2 Background 5

2.1 Consistency Models 5

2.1.1 Sequential Consistency 6

2.1.2 Release Consistency 8

2.2 Hardware Techniques to Enhance the Performance of Consistency Models 10

2.2.1 Hardware prefetching 11

2.2.2 Speculative reads 13

3 Evaluation Methodology 15

3.1 Simulated Architectures 15

3.1.1 Memory System and Network 15

3.1.2 Base Processor 19

3.1.3 Variations on the Base Processor 20

3.2 Simulation Methodology and Environment 22

3.3 Performance Metrics 23

3.4 Applications 24

4 Impact of Hardware Prefetching and Speculative Reads on the Performance of Consistency Models 26

4.1 Sequential Consistency Implementations 27
4.1.1 Prefetching with Write-Through L1 Caches 27
4.1.2 Speculative Reads with Write-Through L1 Caches 33
4.1.3 Impact of Write-Back L1 Caches 35
4.1.4 Summary for SC 36
4.2 Release Consistency Implementations 37
4.2.1 Performance with Write-Through L1 Caches 37
4.2.2 Impact of Write-Back L1 Caches 38
4.2.3 Summary for RC 39
4.3 Comparing SC and RC 39
4.4 Impact of a More Aggressive Protocol 40
4.5 Summary 43

5 Related Work 45

5.1 Consistency Models 45
5.2 Evaluation of Consistency Models 45
5.3 Aggressive Implementations of Sequential Consistency 46
5.4 Prefetching 46

6 Conclusions 48

6.1 Thesis Summary 48
6.2 Future Work 49

Bibliography 51

Illustrations

2.1 Conceptual representation of sequential consistency (SC) 7
2.2 Comparison of straightforward implementations of consistency models 10
2.3 Effect of hardware prefetching and speculative reads with SC 11

3.1 Overview of the simulated system. 16
3.2 Default simulation parameters. 17
3.3 Processor micro-architecture. 19
3.4 Applications used and input sizes 25

4.1 Evaluation of consistency models – Erle, FFT, and LU 28
4.2 Evaluation of consistency models – Mp3d, Radix, and Water 29
4.3 MSHR occupancy – Erle, FFT, and LU 30
4.4 MSHR occupancy – Mp3d, Radix, and Water 31
4.5 Number of rollbacks (with write-through cache) 35
4.6 Effect of aggressive coherence protocol 41

Chapter 1

Introduction

1.1 Motivation

Multiprocessor systems built from commodity high-performance uniprocessors offer a cost-effective solution to achieve high performance. Shared-memory multiprocessor systems, in contrast to the alternative of message-passing systems, provide an abstraction of a single address space that greatly enhances the ease of programming and compiler design on such systems. However, long memory latencies remain a significant impediment to achieving the full performance potential of shared-memory multiprocessors. Current technology trends indicate further increases in the difference between processor and memory speeds [43]. A number of optimizations (caching, buffering, and pipelining) have been proposed in the literature to reduce the impact of long memory latencies in multiprocessor systems. However, these optimizations can potentially cause memory operations to be executed in an order different from that specified by the programmer, sometimes resulting in incorrect executions.

The *memory consistency model* (also called the *memory model* or the *consistency model*) of a shared-memory multiprocessor system is an architectural specification of the order in which memory operations will *appear* to execute to the programmer. The most intuitive memory model considers the multiprocessor as a multiprogrammed uniprocessor. This intuitive memory model is called *sequential consistency* (abbreviated as SC)[23] and guarantees that all memory operations appear to execute one at a time, and that operations of a single processor appear to execute in the order specified by that processor's program. intuitive

Relaxed consistency models relax some of the requirements of memory ordering enforced by sequential consistency. One of the most relaxed models is *release consistency* (abbreviated as RC) [14]. Release consistency requires synchronization accesses in the program to be identified and classified as acquires (e.g., locks) or releases (e.g., unlocks) and allows significant overlap of memory accesses between synchronization points. Release consistency can thus get increased performance at the expense of a more complex programming model.

Previous studies have shown that the release consistency model significantly outperforms the conceptually simpler model of sequential consistency [11, 16, 13, 44], albeit with increased programming complexity [2, 14]. However, the first two of these studies [11, 16] assumed single issue, statically scheduled, previous-generation processors with blocking reads. In contrast, current and next generation high performance microprocessors exploit increased levels of instruction-level parallelism (ILP), using aggressive techniques such as multiple issue, dynamic scheduling, speculative execution, and non-blocking reads. The third study [13] assumed an aggressive current-generation processor, but examined only straightforward implementations of the consistency models. Further, the study used trace-driven simulations which required making significant approximations that were not validated (e.g., constant latency for remote miss). The fourth study [44] examined one optimization with non-blocking reads, but assumed single issue, statically scheduled processors.

Current and next-generation processors that aggressively exploit ILP allow two hardware optimizations, hardware-controlled non-binding prefetching and speculative reads [12], that can enhance the performance of sequential consistency and release consistency. These optimizations have been conjectured to equalize the performance of consistency models. Hardware prefetching and speculative reads [12] have recently begun to appear in commercial microprocessors (e.g. HP PA-8000 [17], Intel Pentium Pro [18], and MIPS R10000 [25]), and re-open the issue of whether the hardware

performance advantages of relaxed consistency models justify the tradeoff in programming complexity.

Furthermore, for earlier processors with blocking reads, the decision to support a relaxed consistency model did not necessarily have to be made at processor design time, since writes can be made non-blocking by simply providing an early acknowledgment from an external memory controller. Non-blocking reads, however, bring in a value needed by other instructions and must be integrated into the processor design, Thus, the consistency model now has a larger impact on processor design, further increasing the importance of understanding the benefits of relaxed consistency on current processors.

1.2 Contributions of this Thesis

This thesis performs the first detailed quantitative comparison of several implementations of sequential consistency (**SC**) and release consistency (**RC**) with processors supporting dynamic scheduling and non-blocking reads.

We use detailed execution-driven simulation of six applications to compare the hardware performance of SC and RC with ILP processors, in both simple implementations as well as with the enhancements provided in current ILP processors (hardware prefetching and speculative reads). Our simulator accurately models the internals of an aggressive ILP processor similar to the MIPS R10000 along with an aggressive memory system. The key results of this study are as follows.

- For SC, the two techniques dramatically improve performance, providing up to a factor of 2 speedup.
- For RC, overall, the two techniques are not very effective, because RC already manages to hide all write latency and a large part of read latency.
- Comparing SC and RC, we find that RC consistently outperforms SC. With write-through primary caches, RC achieves a speedup of over 1.5 for four of

the six applications. With write-back primary caches, the speedups are less dramatic, but still fairly large (1.25 or more for four applications). With a more aggressive, but more complex, cache-coherence protocol, optimized SC achieves performance comparable to RC on two applications, but a significant gap remains for others.

Overall our results show that RC hardware has significant performance benefits over SC hardware, and at the same time requires less system complexity with ILP processors. Write latencies that are not overlapped with SC are mainly responsible for the performance difference between SC and RC.

1.3 Organization

The rest of this thesis* is organized as follows. Chapter 2 presents the background material for this thesis. Chapter 3 discusses our simulated architecture, methodology, and our applications. Chapter 4 describes the results of our comparison of consistency models. Chapter 5 discusses related work. Chapter 6 summarizes the results from this dissertation and discusses future work.

*Large portions of this thesis are based on an earlier work [30] which is copyright by the ACM, 1996.

Chapter 2

Background

This chapter provides the background information for the rest of the thesis. Section 2.1 defines the concept of a consistency model and briefly describes the consistency models that we study in this thesis. Section 2.2 describes hardware prefetching and speculative reads, currently implemented hardware optimizations for consistency models that we study in this work.

2.1 Consistency Models

The *memory consistency model* (also called the *memory model* or the *consistency model*) of a shared-memory system is an architectural specification of the order in which memory operations *appear* to execute to the programmer. The programmer of an explicitly parallel program has to understand the memory consistency model to reason about the results that the program can produce. The system (hardware or compiler) designer, on the other hand, has to understand the memory consistency model in order to implement only those performance improving optimizations that do not violate the constraints of the memory model. The memory consistency model thus forms an integral part of both the system design process and the application code writing process.

On uniprocessor systems, the memory model presented to the programmer usually guarantees that all memory operations appear to execute one at a time and in the order specified in the program. The uniprocessor system includes both hardware and compiler optimizations that improve performance by possibly reordering or overlapping multiple memory operations. However, care is taken to maintain uniprocessor

control and data dependences which ensure that memory operations *appear* to execute in program order.

The memory model in a multiprocessor system is more complicated than that of the uniprocessor system because of the possibility of concurrent reads and writes to the same location by different processors. A number of memory consistency models for multiprocessors have been proposed in the literature. We next briefly discuss the two consistency models that we use in this thesis.

2.1.1 Sequential Consistency

The most intuitive memory model on multiprocessors retains the characteristics of the memory model on uniprocessors and extends it for multiprocessors. A simple definition of this model would require the ordering of memory operations to model that produced by a multiprogrammed uniprocessor. This memory model is called *sequential consistency* (abbreviated as SC) and was formally defined by Lamport [23] as follows:

[A multiprocessor is sequentially consistent if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order and the operations of each individual processor appear in this sequence in the order specified in the program.

The two requirements of sequential consistency therefore are (1) all memory operations appear to execute *atomically* (one at a time) in some order and (2) all memory operations of a processor appear to execute in *program order*. Figure 2.1 shows the conceptual representation of a sequentially consistent system [1]. The multiprocessor system is conceptually equal to multiple processors sharing a single memory module through a central switch.

A naive implementation of sequential consistency would require a processor to wait for a memory operation to complete before issuing its next memory operation in pro-

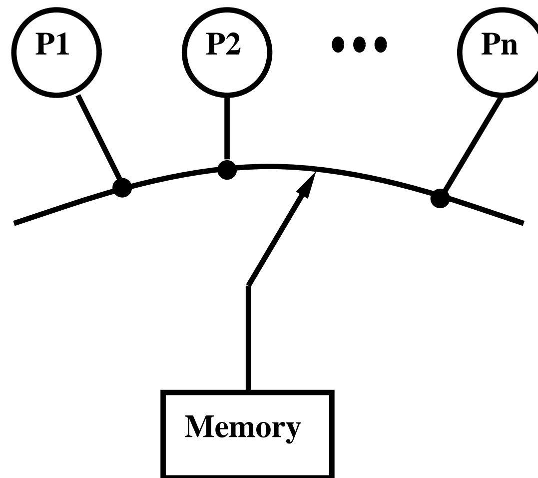


Figure 2.1 Conceptual representation of sequential consistency (SC)

gram order. This requirement, however, can seriously impact the use of performance improving optimizations to reduce the impact of memory latency in multiprocessors. These optimizations (both at the hardware and software levels) use buffering, pipelining and reordering to reduce the impact of memory latency in multiprocessors. Note that the importance of such optimizations is increased in multiprocessors compared to uniprocessors because of the presence of longer (remote) memory latencies and coherence misses.

For example, hardware optimizations like a dynamically scheduled processor (which allows out-of-order execution) or a write-buffer (which allows reads to bypass writes), or an aggressive cache implementation (lockup-free caches), or an aggressive network that does not maintain ordering of requests can all result in violations of the ordering requirements of sequential consistency. Similarly, compiler optimizations like register allocation, code motion, common sub-expression elimination, and loop interchange can potentially reorder (or eliminate) memory operations and violate the requirements of sequential consistency [1]. A straightforward implementation of se-

quential consistency would prevent the use of these optimizations and constrain the processor to execute memory operations one at a time in program order.

However, sequential consistency does not require memory operations to be executed in this way, it only requires that the execution *appear* as if the memory operations were executed one at a time in program order [23]. Many schemes using this observation to enhance the performance of sequential consistency have been proposed in the literature. In this thesis, we evaluate two hardware techniques [12] that are currently implemented in commodity microprocessors to improve the performance of sequential consistency (described in Section 2.2). Other techniques have been proposed to improve the performance of sequential consistency [3, 24, 11]. However, these techniques either require very aggressive networks, complex hardware, or aggressive compiler technology.

2.1.2 Release Consistency

Relaxed memory consistency models allow certain memory operations to execute out of program order or non-atomically, thus enabling more optimizations that require overlap and reordering of memory operations. However, these models require the programmer to be explicitly aware of the effect of such reorderings and write the program suitably to ensure correctness. A number of relaxed consistency models, differing in the way in which they relax the requirements between various classes of memory operations, have been proposed [23, 8, 15, 9, 14, 19, 41, 7]. We next discuss one of the most relaxed models – *release consistency* (abbreviated as RC).

Release consistency exploits the key observation that most programs use synchronizing memory operations to ensure ordering between concurrent accesses to the same memory location by different processors. Release consistency requires that synchronization accesses in the program be identified and classified as either *acquires* or *releases*. Informally, acquires are read synchronization operations that are used to order data operations, and releases are write synchronization operations that are used

to order data operations. Release consistency uses this information to allow great flexibility in reordering (and buffering/pipelining) of data accesses between synchronization points. Release consistency is formally defined as follows [14]:

[A system is release consistent if] (1) before an ordinary read or write access is allowed to perform with respect to any other processor, all previous acquire accesses must be performed, and (2) before a release access is allowed to perform with respect to any other processor, all previous ordinary read and write accesses must be performed, and (3) all special accesses are processor consistent† with respect to one another.

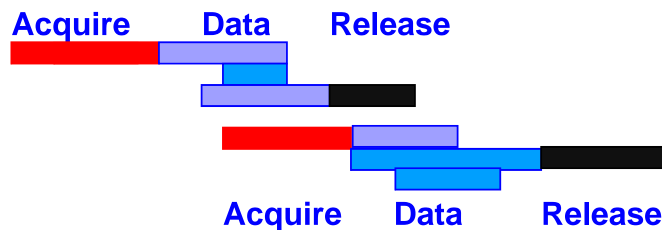
Figure 2.2 graphically demonstrates the difference between straightforward implementations of SC and RC. Figure 2.2(a) shows the execution profile of a straightforward implementation of sequential consistency where memory operations execute one after the other in order and Figure 2.2(b) shows the execution profile of a straightforward implementation of RC. As seen from the figure, the main advantage of release consistency is the potential for increased performance due to increased overlap of memory operations. The disadvantage is the more complex programming model.

Note that the previous definitions of sequential consistency and release consistency only require memory operations to *appear* to execute in the order specified by the memory model. A number of techniques that use this observation to enhance the performance of consistency models have been proposed in the literature [12, 3, 24, 11]. We next discuss two such hardware techniques for ILP processors that can be used to improve the performance of consistency models. The other techniques are either too restrictive, require more aggressive, complex system implementations, or do not appear to give significant performance benefits.

---

†Informally, processor consistency differs from sequential consistency in that it allows reads to bypass writes. A more formal definition can be found in [15, 14].

(a) Sequential consistency

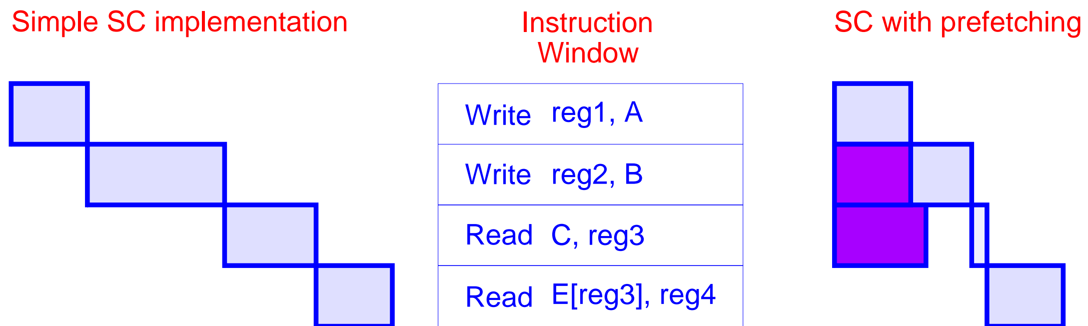


(b) Release consistency

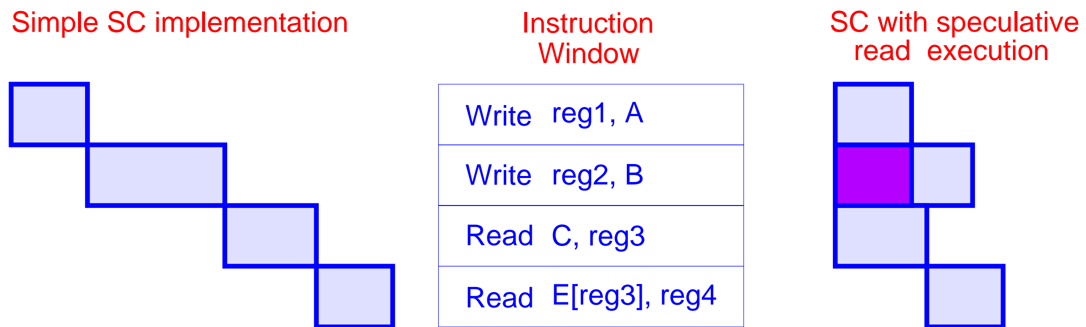
Figure 2.2 Comparison of straightforward implementations of consistency models

2.2 Hardware Techniques to Enhance the Performance of Consistency Models

Current and next-generation processors exploit instruction-level parallelism (ILP) using techniques like multiple instruction issue, dynamic (out-of-order) scheduling, register renaming, speculative execution, and non-blocking reads. The processor exploits ILP by examining a large window of instructions (called the *instruction window* or *active list* [25]) at a time, and executes the instructions that are not dependent on the completion of any previous incomplete instructions. For such processors, Gharachorloo et al. have proposed two hardware techniques to improve the performance of consistency models. These two techniques, hardware prefetching and speculative reads [12], exploit the instruction lookahead window in an aggressive ILP



(a) Effect of hardware prefetching



(b) Effect of speculative reads

Figure 2.3 Effect of hardware prefetching and speculative reads with SC

processor and take effect whenever the constraints of the memory consistency model could restrict the issue of a memory operation.

Section 2.2.1 describes hardware-controlled non-binding prefetching from the instruction window, and Section 2.2.2 describes speculative read execution.

2.2.1 Hardware prefetching

Prefetching seeks to bring the contents of a memory location into the processor cache before the demand access to that location. The hardware prefetch technique issues a hardware-controlled non-binding prefetch [16] for a decoded memory operation in

the instruction window as soon as its address is available, and if the operation cannot be issued otherwise due to consistency constraints. This technique seeks to reduce memory latency by partially servicing large latency accesses that are delayed due to consistency constraints. In addition to the data of the specific prefetched location, prefetches usually bring in extra data, typically other data that is included in one transfer unit of the cache (the cache line size).

A read prefetch can be used to bring the data into read-shared state while the operation is delayed due to consistency requirements. On a sequentially consistent system, read prefetches can be used to obtain remote data for reads while a regular memory operation is pending. On a release consistent system, read prefetches can be used to prefetch reads past acquire operations. Since non-binding prefetches are still visible to coherence actions which make sure that the value is updated, we are guaranteed that the actual demand read will return the correct value.

A write prefetch (also referred to as a read-exclusive prefetch) can be issued to acquire exclusive ownership of the data in addition to fetching the data into the cache. A write prefetch can be issued in SC when the actual write operation is delayed due to consistency requirements. Additionally, with ILP processors, writes cannot be issued till they reach the head of the instruction window, to ensure precise exception [38]. Store prefetches can be used in both SC and RC to initiate ownership requests for all such store operations in the instruction window.

Figure 2.3(a) demonstrates the benefits with hardware prefetching from the instruction window on SC. As the figure shows, hardware prefetching is an effective technique for pipelining large latency references even when the memory consistency model disallows it. A significant amount of the latency seen in the access of locations B and C is hidden by hardware prefetching.

Hardware prefetching, however, fails to boost performance in cases when out-of-order consumption of prefetched values is important, e.g., the read of E in Figure 2.3(a). We next describe a technique that attempts to address this problem.

2.2.2 Speculative reads

Speculative reads [12] extend the benefits of hardware prefetching by speculatively using the values of reads brought into the cache, even while previous demand accesses are incomplete. If a possible violation of memory ordering is detected due to early use of such data, the system rolls back the speculative read and all subsequent instructions.

Speculative read execution preserves correctness by requiring that any data that is speculatively read remain visible to the coherence mechanism. This is achieved by using additional on-chip hardware in the form of a *speculative read buffer* [12]. The speculative read buffer must communicate with the cache, tracking any invalidation, update, or cache replacement operations on cache lines that have had reads issued speculatively to them. If such a message reaches the speculative read buffer, the unit must then interface with the processor's window of active instructions and not only reissue the speculated read, but also roll back all subsequent processor operations. The mechanism used to rollback processor operations after a branch misprediction or an exception recovery can be used to implement this.

As with hardware prefetching from the instruction window, speculative reads can be used with both SC and RC whenever a read operation is not issued because of the constraints of the consistency model. In SC, speculative read execution is used with all reads; in RC, speculative read execution is used with reads past an acquire operation. Figure 2.3(b) demonstrates the benefits of speculative reads used in conjunction with hardware prefetching on an SC system. As seen from the figure, a portion of the latency to access location E can now be hidden with speculative read execution. The ability to consume values speculatively can thus lead to improved performance of the system

Note that both these techniques impose certain basic requirements on the system like the need for a hardware-coherence protocol and non-blocking caches. One important distinction between the prefetch and speculative read techniques is that

the prefetching mechanism is useful even for uniprocessors (to prefetch writes) and can therefore be expected to be employed for purposes other than the consistency model implementation. The speculative read buffer and its data and control paths, however, are required only for aggressive implementations of consistency models and have no application to uniprocessors. Furthermore, these need to be implemented on the processor chip. Both these techniques have recently begun to appear in commercial microprocessors (e.g. HP PA-8000 [17], Intel Pentium Pro [18], and MIPS R10000 [25]).

The remaining chapters in the thesis describe our experimental methodology and present our results evaluating the performance of consistency models with hardware prefetching and speculative reads.

Chapter 3

Evaluation Methodology

This chapter describes the evaluation methodology used in this thesis. Section 3.1 describes the simulated architecture and Section 3.2 describes our simulation environment. Section 3.3 describes the performance metrics we use in this thesis. Section 3.4 describes the applications we use.

3.1 Simulated Architectures

This section describes our simulated architecture. Section 3.1.1 first describes our multiprocessor system including the memory system and network. Sections 3.1.2 and 3.1.3 describe our base processor and the variations to the base processor that we simulate.

3.1.1 Memory System and Network

We simulate a hardware CC-NUMA (cache-coherent non-uniform memory access) multiprocessor, where processing nodes are connected with a two-dimensional mesh network. We simulate an 8-processor system. The base system is illustrated in Figure 3.1. Each processing node consists of a processor, two levels of cache, and a part of the main memory and directory. A split-transaction system bus connects the memory, the network interface and the rest of the system node. The system uses a fully-mapped, invalidation-based, three-state directory coherence protocol.

In our cache hierarchy, the first level is always dual-ported, but can be either write-through with no-write-allocate or write-back with write-allocate. We evaluate both first-level cache configurations since current systems support both kinds of

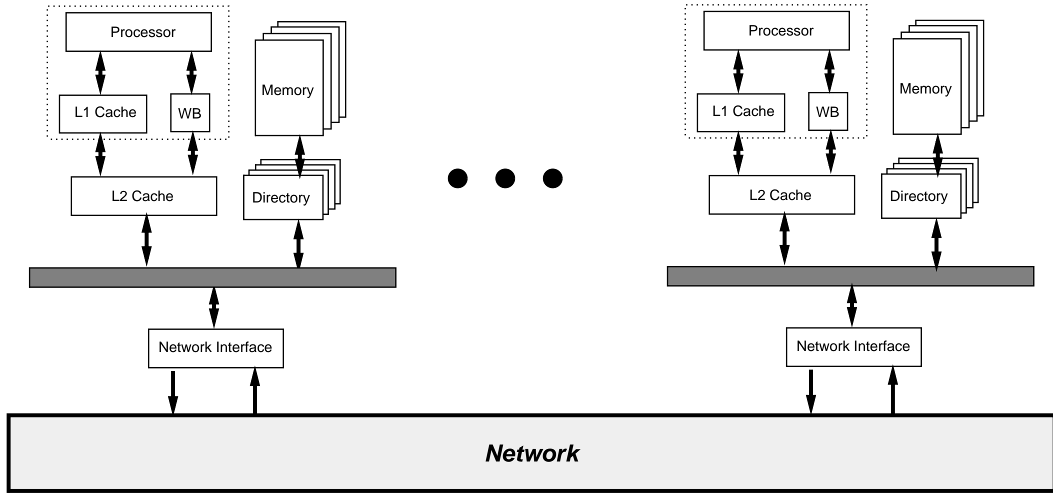


Figure 3.1 Overview of the simulated system.

caches [10, 17, 25, 18]. Additionally, write-hits in SC expose the second-level access latency in a write-through configuration, but only the first-level access in a write-back configuration. In the absence of resource constraints, RC will hide the latency of writes in either configuration. Thus, we expect the comparative results of SC and RC to differ with these different first-level cache configurations. If we have a write-through first-level cache, we also include a coalescing line write-buffer between the two levels of cache. Regardless of first-level cache configuration, the second-level cache is always a pipelined write-back, write-allocate cache. Both the first-level and second-level caches allocate a line for a miss (and possibly evict another line) only upon reply for that line.

Both levels of cache are non-blocking with 8 Miss Status Holding Registers (MSHRs) [22] each. The MSHRs store information about misses and coalesce multiple requests to the same cache line. A maximum of 16 requests can be coalesced in one MSHR. When there is a write request to a line which has a read pending, the MSHR buffers the write and issues an ownership request only when the read reply returns, as in many current processor implementations. We refer to such stalls as

Memory Hierarchy Parameters	
Cache line size	64 bytes
L1 cache (on-chip)	Direct mapped 16K
L1 cache ports	2
L1 MSHRs	8
L2 cache (off-chip)	4-way associative 64K
L2 MSHRs	8
Write buffer (coalescing)	8 line entries
Maximum coalescions per line	16
Memory interleaving	4-way
Memory Latency Components	
L1 cache access	1 cycle
L2 cache access	8 cycles
Memory bus arbitration delay	3 cycles
Directory and memory access	18 cycles
Memory transfer bandwidth	16 bytes/cycle
Network Parameters	
Network speed	150MHz
Network width	64 bits
Flit delay (per hop)	2 network cycles
Bus type	Split-transaction
Bus speed	100 MHz
Bus width	128 bits
Processor Parameters	
Processor Speed	300MHz
Peak issue, retire rate	4 instructions/cycle
Instruction window size	64
Memory queue size	32
Functional units	2 integer arithmetic
	2 floating point
	2 address generation
Branch speculation depth	8

Figure 3.2 Default simulation parameters.

write-after-read stalls. In addition to preventing overlap of the store ownership request with load latency, our implementation of write-after-read stalls also blocks a cache port, possibly preventing later requests from issuing to the cache. Although our system is representative of current systems, this decision can potentially affect the performance of write prefetching. Allowing this ownership request to overlap with a previous read request increases the complexity at the directory controller and at the MSHRs, since potential reordering of requests in the network will now need to be handled by the system. Section 4.4 assesses the impact of the more aggressive and more complex protocol where the read and ownership requests are overlapped.

Figure 3.2 gives our default primary memory system parameters. We have chosen smaller cache sizes than commercial systems, commensurate with our application input sizes (described in Section 3.4) and following the working-set evaluations of Woo et al. [42]. Our secondary cache sizes are chosen such that secondary working sets of most of our applications do not fit in cache; we choose primary cache sizes such that any applications with fixed-size primary working sets fit in cache.

Main memory is 4-way interleaved (by cache line) and is accessed through a pipelined split-transaction bus. The interconnect network is a wormhole-routed two-dimensional mesh network. The system uses separate reply and request networks for deadlock prevention. Parameters common to all the systems are given in figure 3.2. These values assume a 300MHz processor with a 3ns on-chip L1 SRAM cache and a 30ns L2 SRAM cache. The network is assumed to be a 150MHz, 8 byte wide network with 70ns DRAM memory and 4-way interleaved directories. The processor, network, and base memory system parameters are fairly aggressive, and meant to represent current aggressive implementations. The parameters were chosen by extrapolating from numbers given by various system vendors.

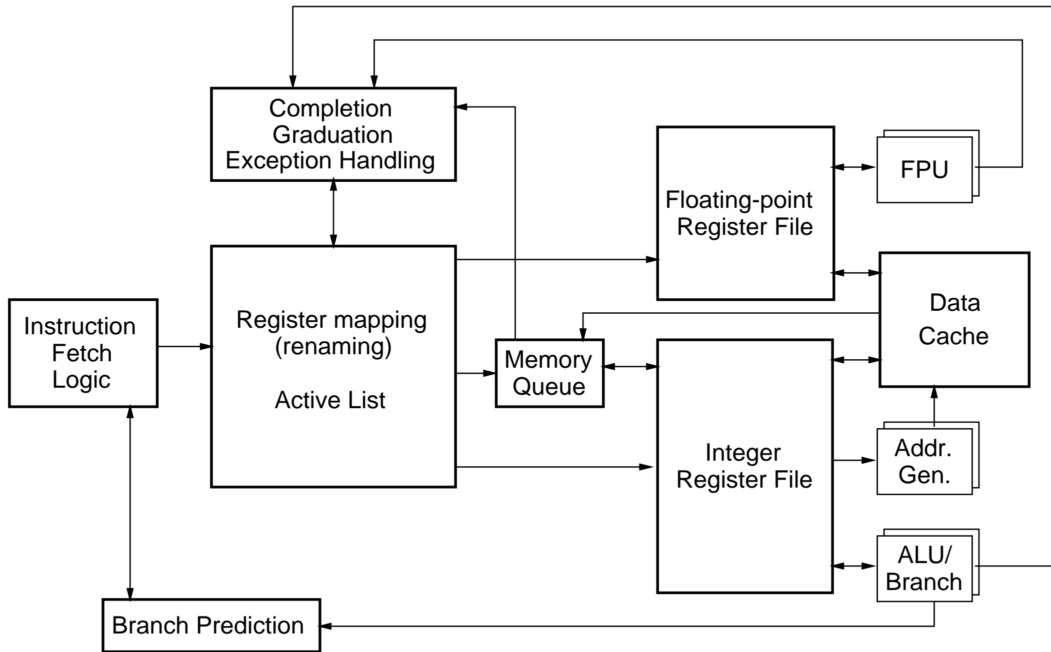


Figure 3.3 Processor micro-architecture.

3.1.2 Base Processor

Figure 3.3 shows the processor micro-architecture that we model. Our base processor model employs widely used techniques to exploit instruction-level parallelism, such as multiple instruction issue, dynamic (out-of-order) scheduling, register renaming, speculative execution, and non-blocking reads. The processor exploits ILP by examining a large window of instructions (called *instruction window* or *active list* [25]) at a time, and executes the instructions that are are ready to be issued, even before the completion of any previous incomplete instructions. This allows instructions to issue and complete out of program order. In particular, the pipeline stages corresponding to Fetch, Decode, and Graduation occur in order, while Issue, Execution, and Completion proceed out of order. Except for writes in release consistent systems, an instruction retires (graduates [25]) when it is complete and when all preceding instructions (by program order) have retired. A write in a release consistent system

retires when its address and value are resolved, and when all previous instructions have retired. To guarantee precise interrupts [38], writes are not issued into the memory system until they reach the head of the instruction window. We use the SPARC V9 `MEMBAR` [39] instructions (memory barriers or memory fences) to enforce ordering of memory operations as required by the consistency model.

The processor also uses a two-bit hardware branch prediction scheme to enable speculative execution past branches. A maximum of 8 speculation levels is supported in the processor. Our base processor uses the notion of *shadow mapping* [25] (similar to the MIPS R10000 and the HP-PA8000) to restore the state of a system on a branch misprediction. Additionally, the processor also supports *soft exceptions* [25] to rollback execution on an internally-generated exception condition.

The processor micro-architecture is most closely based on the MIPS R10000 design [25] though it incorporates features from several other commercial microprocessors like the HP-PA800 [17] and the Intel Pentium Pro [18]. Figure 3.2 gives the processor parameters used in our simulations. These parameters were chosen to model next-generation aggressive processors. The default latencies for the various execution units approximate those for the UltraSPARC [40].

3.1.3 Variations on the Base Processor

The base processor model directly supports the simple implementation of release consistency (RC). Variations on our processor and memory system include a sequentially consistent (SC) processor model, support for hardware-controlled non-binding prefetching, and support for speculative read execution.

For a simple implementation of SC, we modify RC's aggressive base memory system to issue a memory operation only when the previous memory operations of that processor have completed. This method maintains ordering of all memory operations as required by SC. Furthermore, a write in SC does not retire from the instruction

window until it is globally performed [35]. Unlike the RC model which can retire up to 4 writes a cycle, the SC model can retire only at most 1 write per cycle.

To implement hardware prefetching, we issue prefetch requests to the cache as described in Section 2.2. We prefetch requests to the level of cache appropriate for the corresponding demand fetch; thus write prefetches with the write-back write-allocate primary cache and all read prefetches go to the primary cache. Write prefetches for lines not present in the write-through non-write-allocate primary cache only fetch into the secondary cache; bringing these to the primary cache would defeat the purpose of a no-write-allocate cache.‡ Read prefetches always fetch the line to the first-level cache. Prefetch instructions are removed from the memory queue once they are issued to the memory system. Prefetches that cannot be issued due to resource contention are dropped. This model assumes that the increase in the stall time due to resource contention outweighs the reduction in the memory latency due to prefetches, and has been used in previous studies with indiscriminate prefetches [26].§

We implement the speculative read buffer at the processor and use a mechanism similar to but more optimistic than the soft exception mechanism employed by the MIPS R10000 to rollback execution as described in Section 2.2.2. In SC systems, we use the speculative read buffer whenever we want to issue a read out of order; in RC systems, this buffer is only used past acquires. We do not impose a constraint on the size of the speculative read buffer, limiting it only by the number of reads in the memory queue (32 in our base configuration). We force a rollback when the primary cache gets a coherence request from an external source or an invalidation request

‡We performed experiments with an alternative strategy that brought write prefetches to the first-level write-through cache. For all our applications, the performance was comparable to, or less than that with the default strategy of prefetching writes only to the second-level cache.

§We performed experiments varying the prefetch drop strategy for our applications. The alternate prefetch strategy keeps prefetches in the memory queue and issues them only when resources free up. With three of our applications (LU, Mp3d, and Water), SC gets a reduction in execution time between 8% and 12%. Mp3d experiences a slowdown of 8% on RC with the alternate prefetch drop strategy. All other applications exhibit marginal changes to performance with change in the prefetch strategy.

from the secondary cache for inclusion; there is no need to rollback on primary cache replacements since those lines will still remain visible to external coherence. We assume a zero cycle recovery penalty (the number of cycles that are taken to flush subsequent instructions from the instruction window). Our results show that, on our base system, the number of rollbacks is very small, and thus the rollback penalty does not significantly impact the performance of the optimized models.

3.2 Simulation Methodology and Environment

We have developed the Rice Simulator for ILP-based Multiprocessors (RSIM) to model the architecture described in Section 3.1 [28]. In contrast to many current direct-execution simulators, RSIM models both the processor pipelines and the memory subsystem in detail, including contention at various resources. The code for the memory system and network is heavily drawn from RPPT (the Rice Parallel Processing Testbed) [5, 31]. RSIM is execution-driven; i.e., it is driven by application executables rather than traces so that interactions between the processors during the simulation can affect the course of the simulation. The detail in our simulator thus leads to increased simulation times compared to those seen in either direct-execution or trace-driven simulations; however, the detail is necessary for the problems addressed in this thesis.

The applications are compiled with a version of SPARC V9 gcc modified to eliminate branch delay slots¶ and restricted to 32 bit code, optimized with `-O2 -funrollloop`. The resulting binary is fed into a predecoder which expands the instructions into a format that can be interpreted by our simulator.

To speed up the simulations, we assume all instructions hit in the instruction cache (with 1 cycle hit time) and private (i.e., non-shared) variables also hit in the data cache. Both of these approximations have been widely used in shared-memory

¶RSIM does not currently support architected delay slots.

multiprocessor performance studies. However, we do model contention due to private data accesses at various processor and cache resources.

3.3 Performance Metrics

We use the execution time as the primary metric to evaluate performance. We also divide execution times into its various components, namely CPU time, data memory time, and synchronization time. However, with ILP processors, each instruction can potentially overlap its execution with both previous and following instructions. Hence, it is difficult to assign stall time to specific instructions. We adopt the following convention also used in other work [34]. We count a cycle as part of busy time if we retire the maximum number of instructions possible in that cycle (four in our system). Otherwise, we charge that cycle to the stall time component corresponding to the first instruction that could not retire in that cycle. Thus, effectively, the stall component for a class of instruction represents the cumulative time that instructions in the class stall at the top of the instruction window before retiring. If an instruction retires without having spent any stall time at the top of the instruction window, it is considered to have fully overlapped with previous instructions. We use these detailed statistics only to gain insight into the nature of the various applications and to identify the portions of the computation overlapped by various optimizations. For purposes of comparing various implementations, however, we use the total execution time as the primary performance metric. We additionally subdivide the data memory stall time into the time spent on L1 hits, L2 hits, local memory accesses, and remote memory accesses, for both reads and writes. Henceforth, we use the term *memory stall time* to denote the data memory stall component of execution time.

We also use statistics on MSHR occupancies to give us an idea of the usage of MSHRs in the L1 and L2 caches and the extent to which memory operations are overlapped in our systems.

3.4 Applications

We use six applications in this study – Radix, FFT and LU from the SPLASH-2 suite [42]; Water and MP3D from the SPLASH suite [37]; and Erlebacher, obtained from the Rice parallel Fortran compiler group [4]. We describe the applications briefly below.

LU is a non-contiguous version of the kernel from the SPLASH-2 suite modified to use flags instead of barriers to improve performance. The version of LU that we use also includes loop transformations and procedure inlining optimizations to better exploit the ILP features in the processor [29]. **FFT** from the SPLASH-2 suite performs a 1D Fast Fourier Transform using a six-step algorithm. The version of FFT that we use also includes ILP-specific code transformations to improve performance [29]. The majority of communication in this application occurs in the transpose phase. **Radix** is an integer sorting kernel from the SPLASH-2 suite. **Mp3d** is an application from the SPLASH suite performing a Monte Carlo simulation of a rarefied flow simulation. **Water** is an N-body molecular dynamics simulation from SPLASH. **Erlebacher** solves partial differential equations by performing 3-D vectorized tridiagonal solves using the Alternating-Direction-Implicit (ADI) method [4]. The key data structures are 3-dimensional arrays which are distributed by assigning a consecutive block of X-Y planes to each processor. One phase dominates the execution time, and contains all the communication and synchronization of this application. The computation in this phase consists of a forward-substitution pipeline and a backward-substitution pipeline, with flags to synchronize processors sharing a boundary plane. The block size determines the size of each pipeline stage.

The input sizes for our applications run on an 8-processor system are summarized in Figure 3.4. These are greater than or equal to the sizes used in the SPLASH and SPLASH-2 distributions for all applications except LU. In the case of LU, owing to higher simulation times with our detailed simulator, we use a problem size one step smaller than recommended, but do not expect the results to be affected since we run

Application	Input Size
LU	256 by 256 matrix, block 8
FFT	65536 points
MP3D	50000 particles
Water	512 molecules
Radix	1024 radix, 512K keys, 512K max key
Erlebacher	64 by 64 by 64 cube, block 8

Figure 3.4 Applications used and input sizes

the smaller problem size only on our 8-processor system (the default problem size is recommended for up to 64 processors). We use a first-level 16K cache and a second-level 64K cache. These cache sizes were chosen commensurate with the input sizes of our applications, based on the methodology described by Woo et al. [42]. The primary working set fits in the L1 cache, as these are either input-independent or user-defined in each of our applications. The secondary working sets of most applications (which scale with problem size) do not fit in the L2 cache.

Chapter 4

Impact of Hardware Prefetching and Speculative Reads on the Performance of Consistency Models

This chapter presents our experimental results evaluating the impact of hardware prefetching and speculative reads on the performance of consistency models. Section 4.1 and 4.2 discuss the effect of hardware prefetching and speculative read execution on SC and RC respectively. Section 4.3 compares SC and RC. Section 4.4 examines the impact of a more aggressive coherence protocol. Finally, Section 4.5 summarizes the results of this chapter.

Figures 4.1 and 4.2 summarize our results. As motivated by Section 3.1.1, we investigate systems with write-through and write-back first-level caches (shown on the left and right side respectively of each figure). For each of SC and RC, and for each first-level cache configuration, we examine three systems: `plain` refers to the simple implementation, `+PF` adds read and write prefetching as discussed in Section 3.1.2, and `+SR` further adds speculative reads to the `+PF` configuration. For each implementation, Figures 4.1(a) and 4.2(a) show the total execution time, normalized to the time for a simple implementation of SC using the write-through L1 cache. The execution times are divided into three components – CPU, data memory stall and synchronization stall. Recall that, as described in Section 3.3, the value of each component represents time stalled at the top of the instruction window. Thus, the low CPU times in Figures 4.1(a) and 4.2(a) do not generally imply poor speedup; rather, these values indicate that a large part of the CPU time is completely overlapped with previous longer latency operations. Figures 4.1(b) and 4.2(b) magnify the memory regions of Figures 4.1(a) and 4.2(a) providing a more detailed characterization of memory

stall time. Each bar showing memory stall time is separated into write and read components by a horizontal dividing line. Read and write stall times are further divided into time spent stalled on L1 cache hits, L2 cache hits, misses to local memory, and misses to remote memory.

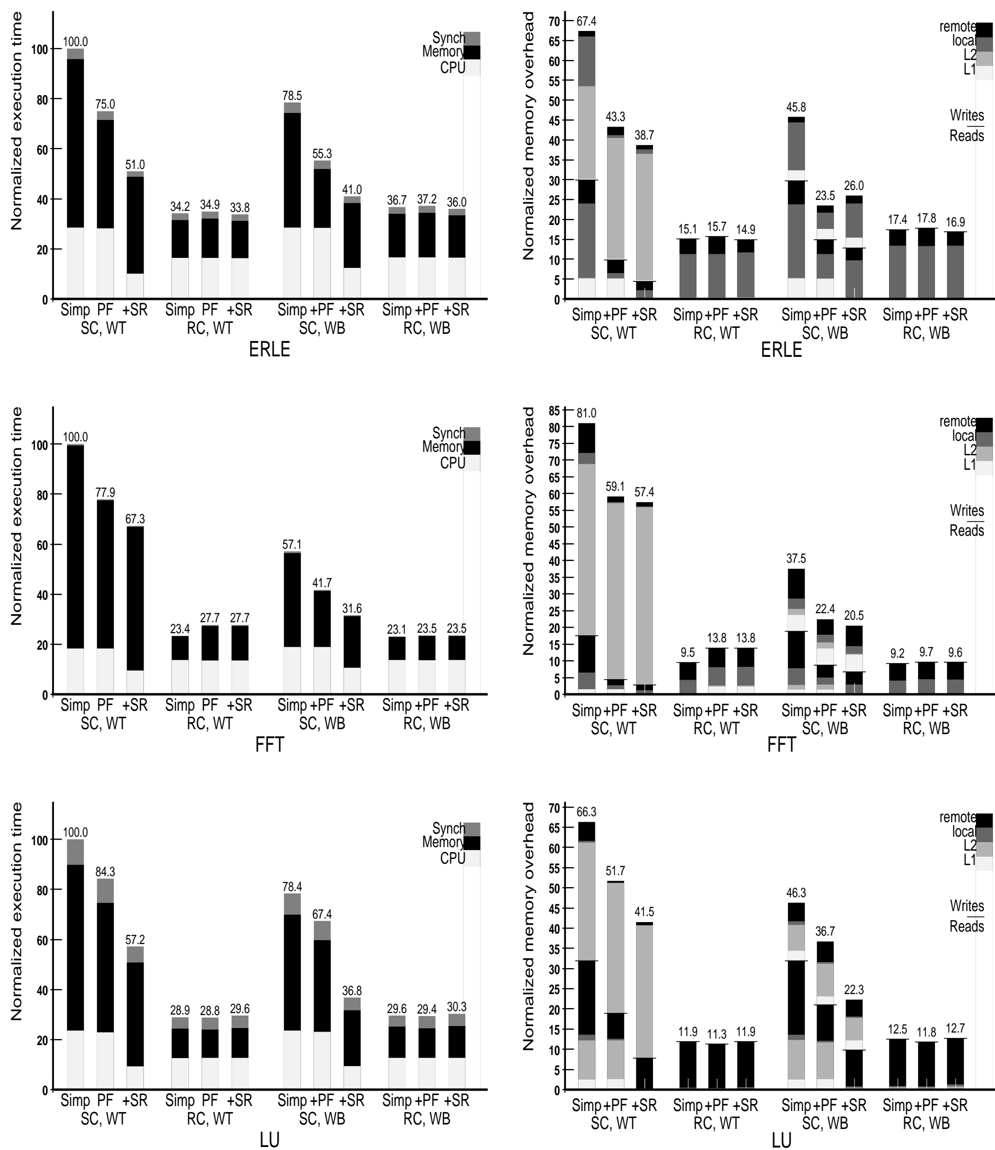
Figures 4.3 and 4.4 provide additional data to indicate the overlap of memory operations in the various implementations. These figures illustrate MSHR occupancy distributions at the L1 and L2 caches respectively for the system with first level write-through caches. They give the fraction of total time (on the vertical axis) for which at least N MSHRs are occupied by misses, where N is the number on the horizontal axis. Recall that only read misses reserve L1 MSHRs, as the L1 write-through cache is no-write-allocate. Both read and write misses reserve L2 MSHRS. The L1 MSHR occupancy graph thus gives an idea of the overlap of read operations and the difference between the L2 and L1 MSHR occupancy graphs indicates the amount of overlap of write operations.

4.1 Sequential Consistency Implementations

Sections 4.1.1 and 4.1.2 analyze the performance impact of hardware prefetching and speculative reads in an SC system with write-through L1 caches. Section 4.1.2 summarizes the results with write-through L1 caches. Section 4.1.3 discusses the impact of a write-back L1 cache.

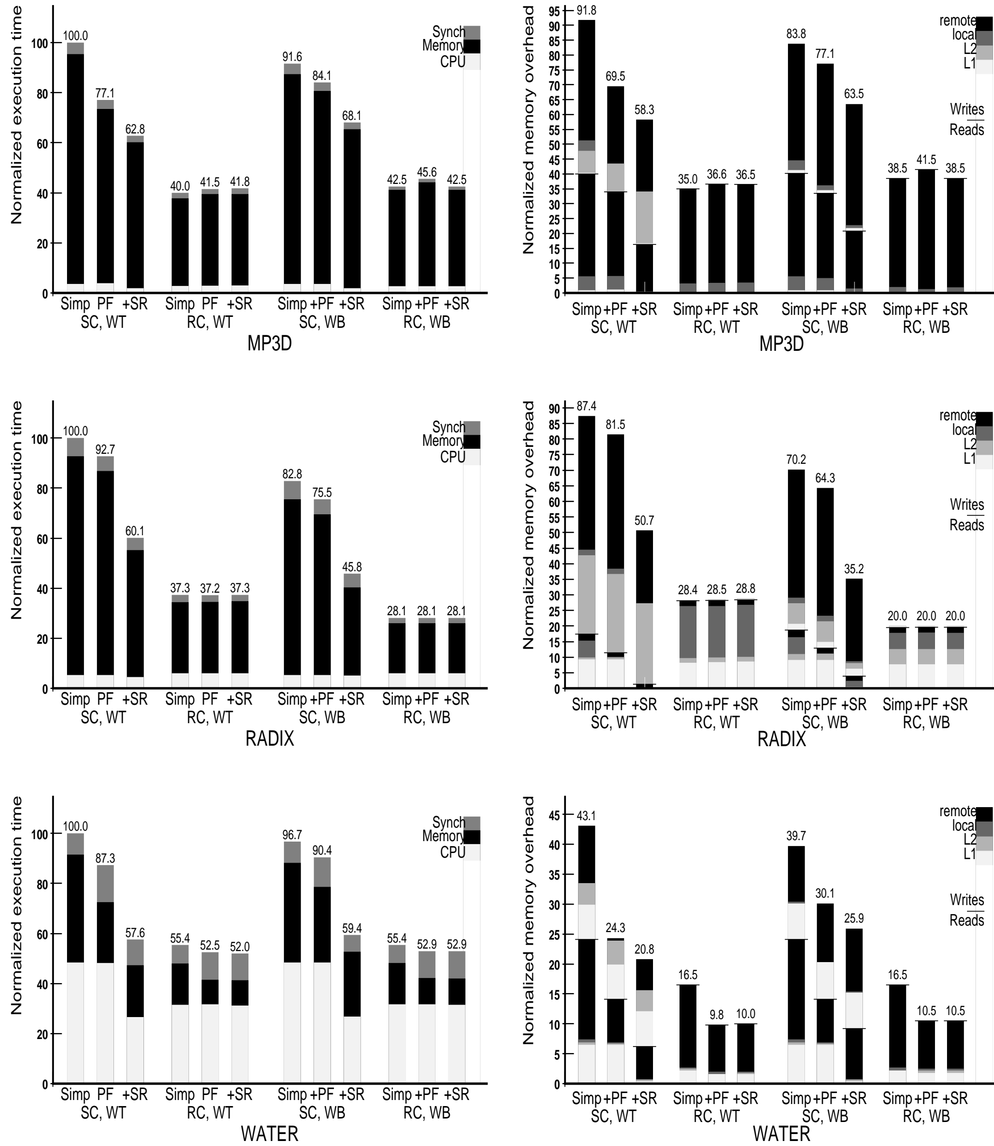
4.1.1 Prefetching with Write-Through L1 Caches

With a write-through cache, hardware-controlled prefetching helps SC performance for all applications, but to a variable extent. All applications experience improvements in execution time ranging from 7% to 23%. Overall, most of the benefits of prefetching appear from reducing read stall time; prefetching is generally unsuccessful in reducing write stall time.



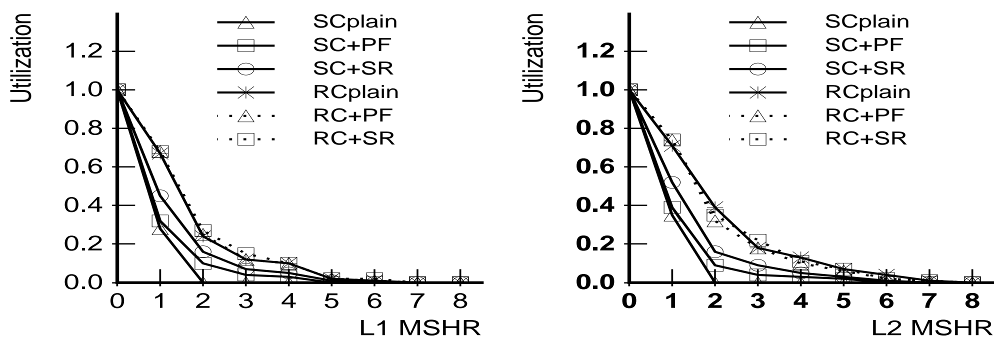
(a) Execution time components (b) Memory time components

Figure 4.1 Evaluation of consistency models – Erle, FFT, and LU

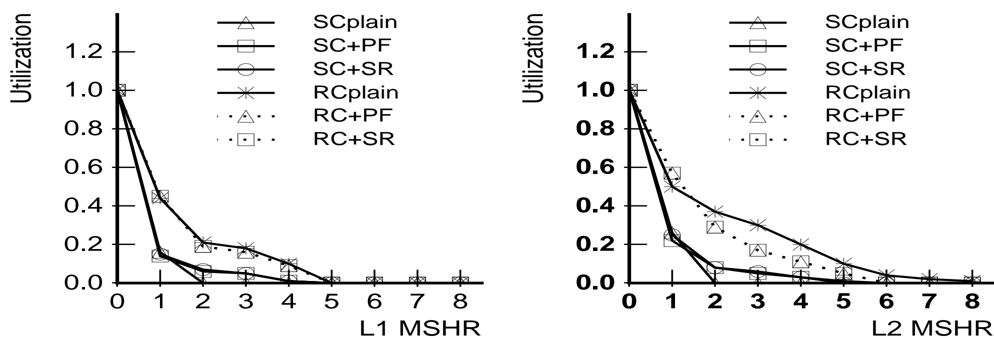


(a) Execution time components (b) Memory time components

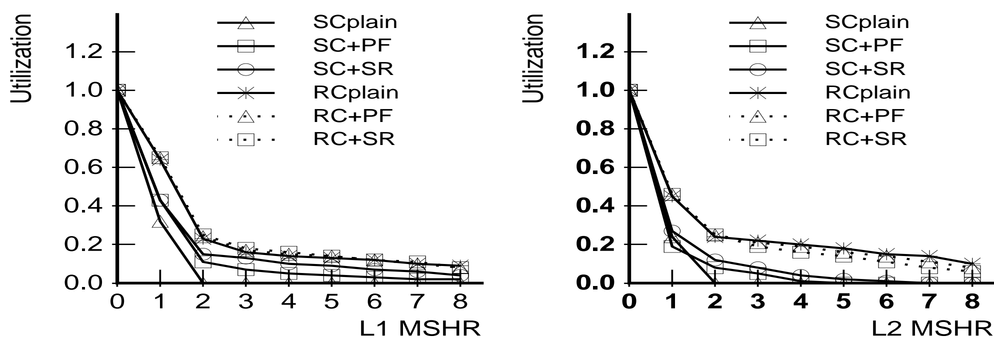
Figure 4.2 Evaluation of consistency models – Mp3d, Radix, and Water



ERLE



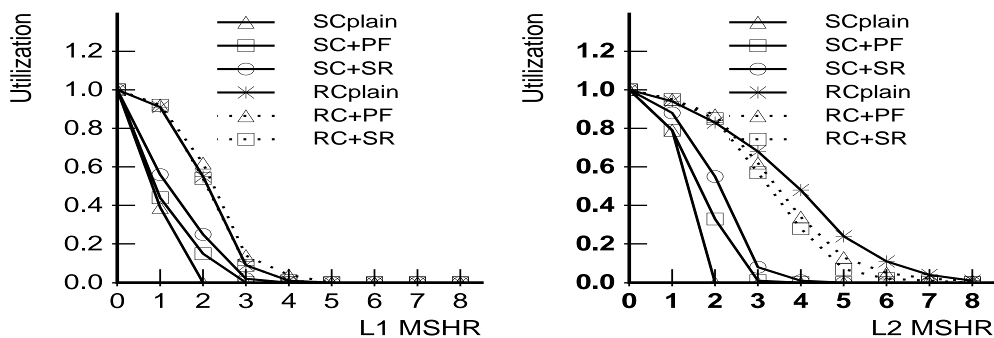
FFT



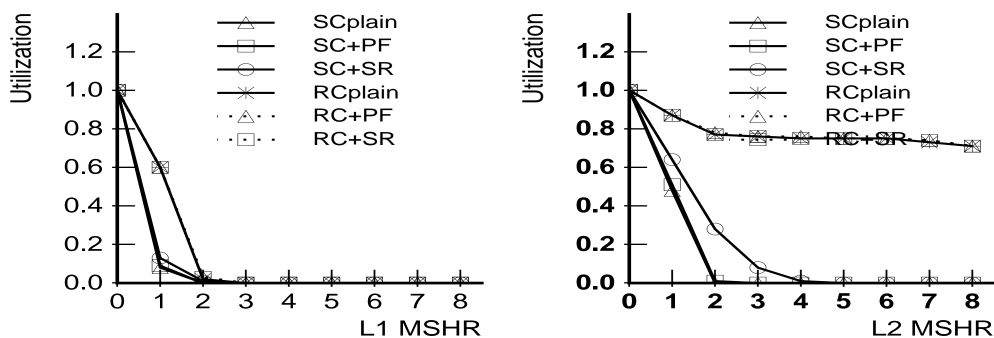
LU

(a) L1 MSHR occupancy (b) L2 MSHR occupancy

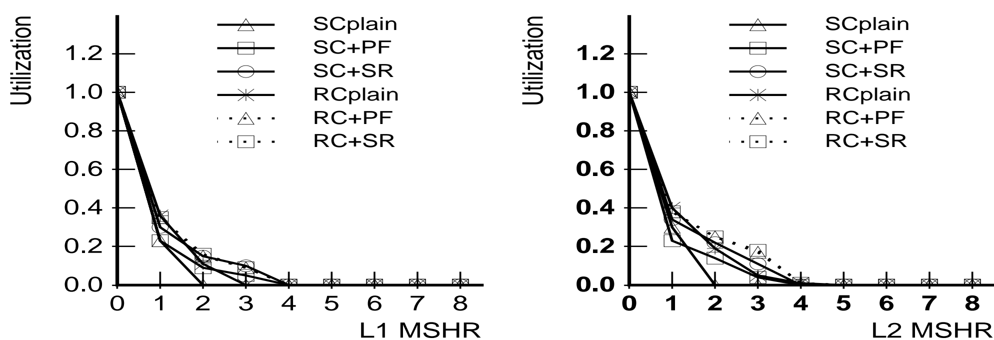
Figure 4.3 MSHR occupancy – Erle, FFT, and LU



MP3D



RADIX



WATER

(a) L1 MSHR occupancy (b) L2 MSHR occupancy

Figure 4.4 MSHR occupancy – Mp3d, Radix, and Water

Several factors limit the benefits of hardware-controlled prefetching in our applications. We next discuss these briefly below.

First, the finite size of the instruction window limits how early a prefetch can be issued. In particular, if there are no other long latency operations before the prefetched instruction in the instruction window, the potential for overlap is limited. Thus, prefetching is most effective when several memory misses occur close together within the instruction window (we use the MSHR occupancy graphs in Figures 4.3 and 4.4 as a measure of *miss clustering* leading to higher overlap).

Second, the address of the instruction to be prefetched may depend on the value of a read instruction that is also blocked from issuing. Since the value cannot be used until the read completes, the later memory operation may not be prefetched early enough. (This is similar to the example discussed in Section 2.2.1.)

Third, for SC with write-through caches, all writes must propagate to at least the L2 cache before they are considered complete, and before they retire from the instruction window. This minimum write latency cannot be overlapped by prefetching.

Fourth, as explained in Section 3.1.1, in our default system, if a write prefetch is issued while a demand or prefetch read to the same cache line is outstanding, the ownership request for the prefetch is blocked until the outstanding read returns. This can further limit the effectiveness of write prefetching.

Finally, prefetching does not result in reduced latency if the prefetched line is either invalidated, replaced, or downgraded to read-only state (for write prefetches) before the corresponding demand access. Such prefetches are called *early prefetches*. Such early prefetches may actually degrade performance by replacing a line that will be used, or invalidating a line that is yet to be used by the remote processor, or increasing the system contention increasing the latency of all accesses.

One or more of the above effects is seen in all our applications. As Figures 4.3 and 4.4 show, most of our applications exhibit greater potential for read miss overlap compared to write miss overlap. Subsequently, most of the benefits from

prefetching appear from reducing read stall time. Mp3d and Water observe some strided write misses and subsequently see some reductions in write stall time as well. Prefetching in Radix and Mp3d is limited due to memory references whose addresses depend on values returned from previous reads (in the permutation phase in Radix, and during cell array accesses in Mp3d). Additionally, all our applications experience L2 write latencies since all writes must access the secondary cache with a first level write-through cache configuration. Finally, Erle, LU, Mp3d, and Water also experience increased stall times due to write-after-read stalls as discussed above. Mp3d also sees some performance degradation due to early prefetches occurring from high false sharing and races.

4.1.2 Speculative Reads with Write-Through L1 Caches

The addition of speculative read execution helps every application significantly, with improvements in execution time (relative to simple SC) ranging from 23% to 49%. These improvements are seen in CPU time, read stall time, and write stall time.

Compared to SC with prefetching alone, CPU time decreases significantly for all applications making a significant impact on execution time for FFT, LU, Erlebacher, and Water. Each of these applications benefit from the ability to consume the values of reads speculatively, as this ability allows computation dependent on those reads to be largely overlapped with a long latency memory access stalled at the head of the instruction window.

Compared to SC with prefetching alone, all applications see a reduction in read stall time with the addition of speculative read execution. Erle, LU, Radix, and Water see reductions in read stall time mainly due to reductions in the stall time due to L1 hits. Radix, Mp3d, and LU see reductions in other components of read stall time as well. Radix and Mp3d have reads with addresses dependent on the values of previous reads, and subsequently benefit from the ability to consume the value of reads as soon as possible, since this gives them more potential for further overlap. LU

sees a decrease in conflict misses caused by reordering of accesses; in either simple SC or SC with prefetching, LU experiences repeated conflict misses among subsequent demand read accesses, since these must occur in order (most of these conflicts occur at the first-level direct mapped cache; relatively fewer conflicts occur at the second-level four-way associative cache). With speculative read execution, reads can issue in parallel and out of order; as a result, several reads to the conflicting cache lines can occur concurrently. Read misses to the same cache line can coalesce into the same MSHR, while other reads can hit a line despite the fact that its set has a pending MSHR. This eliminates some of the conflict misses seen using a system (such as SC or SC with prefetching alone) in which demand reads can only occur one-at-a-time and in order.

Among all the applications, only Radix sees a reduction in write stall time. Radix sees this additional write stall time reduction not seen earlier with the addition of only prefetching because it also has writes whose addresses depend on values returned by previous long latency reads. The ability to consume values for reads and use those values in sending out prefetches for the above writes improves the write overlap for Radix.

Effect of rollbacks. One potential limitation of speculative execution is that overly optimistic speculation could lead to excessive rollbacks, which may hurt performance (in cases analogous to early prefetching). Recall that we rollback execution whenever we detect an event that signals a possible violation of the memory ordering required by the consistency model – coherence messages and L2 cache replacements in our system.

The first column of Figure 4.5 summarizes the average number of rollbacks seen by each processor in the `SC+SR` system with a first-level write-through cache. The numbers in parentheses indicate the percentage of these rollbacks that are caused due to replacements. As seen in the figure, all our applications see a very small number of rollbacks. Only LU sees more than 1000 rollbacks per processor, and even there, fewer

than 0.2% of the total number of reads cause rollbacks; rollback penalties make up less than 0.05% of total execution time. Additionally, on all our applications except Mp3d, the rollbacks are mainly caused due to coherence messages and not due to replacements.

Summary for Write-Through L1 Caches

In summary, for sequentially-consistent systems with a first-level write-through cache, we find that hardware-controlled prefetching alone improves performance but the improvements are small for some applications; the addition of speculative read execution consistently and significantly increases system performance showing up to a factor of two speedup. Nevertheless, we find that for an architecture with write-through L1 caches, neither technique is sufficient to handle the large write latency component associated with SC. We next determine the possible benefits of using a first-level write-back cache instead.

4.1.3 Impact of Write-Back L1 Caches

Figures 4.1 and 4.2 show that the primary change from a write-through to a write-back L1 cache is in the decreased contribution of write latency to execution time. In Erlebacher, FFT, LU, and Radix, the relative contribution of write latency decreases

Application	Total Rollbacks (% replacement-related)	
	SC+SR	RC+SR
Erle	40.2 (0.0%)	3.6 (0.0%)
FFT	80.5 (0%)	0 (0.0%)
LU	1273.0 (0.1%)	1226.0 (0.1%)
Mp3d	411.5 (12.2%)	1.4 (9.1%)
Radix	582.8 (0.2%)	0 (0.0%)
Water	156.1 (0.7%)	9.4 (0.0%)

Figure 4.5 Number of rollbacks (with write-through cache)

significantly, since many writes that hit in the write-through L1 cache had to experience L2 cache latency; with a write-back L1 cache, these writes must only take L1 access time. In contrast, the write latency component does not drop much in Water and Mp3d. In each of these applications, the write stall component is dominated by remote write misses, on which write-back caches have little effect.

The overall benefits of the two techniques of hardware prefetching and speculative reads on SC systems with write-back L1 caches are qualitatively similar to those with write-through L1 caches. The improvements in execution time range from 7% (Radix) to 30% (Erlebacher) for hardware prefetching, and 26% (Mp3d) to 58% (LU) for speculative read execution (relative to simple SC). As with the write-through case, on all the applications, the two techniques are more successful at reducing read latency rather than write latency. The main reason for the different performance improvements with the optimizations in the write-through and write-back cases is that the write latency forms a smaller part of the execution time in the write-back case.

4.1.4 Summary for SC

For sequentially consistent systems with write-through L1 caches, hardware prefetching alone improves the performance, but the improvements are small for some applications. The addition of speculative read execution consistently and significantly increases system performance showing up to a factor of two speedup. Write back L1 caches additionally reduce a significant portion of the write latency component in many of the applications. However, a significant portion of memory stall time associated with both reads and writes still remains.

4.2 Release Consistency Implementations

We first discuss the impact of hardware prefetching and speculative reads in an RC system with write-through L1 caches (Section 4.2.1) and then discuss their impact on an RC system with write-back L1 caches (Section 4.2.2).

4.2.1 Performance with Write-Through L1 Caches

Figures 4.1 and 4.2 show that, unlike SC, RC does not experience much benefit from hardware prefetching and speculative reads on a system with write-through caches. In fact, these optimizations cause a slowdown on some of our applications.

Qualitatively, there are two key differences in the way the optimizations affect performance with RC and SC. As explained in Section 2.1, RC already allows increased read and write overlap compared to SC; the optimizations help RC only when there is an outstanding acquire or when a write with a known address is waiting to reach the head of the instruction window. Furthermore, once a write reaches the head of the window, it retires immediately. This implies that most write latency is already hidden. Therefore, the net effect on performance of write prefetching is expected to be limited.

Additionally, hardware prefetching from the instruction window can actually hurt performance in two ways. First, write prefetching and prefetching reads and writes past an acquire can result in bringing in data too early, subsequently increasing the network traffic of the system. Second, as seen in Section 4.1, hardware prefetching can result in write-after-read stalls not seen in the base system. These stalls occur as a result of write prefetches being issued for an outstanding read access. The base case issues such writes only after the read completes, and hence does not see the effect of such write-after-read stalls. While both these effects are also seen in SC, the increase in performance due to the reduction in the write miss stall time helps hide the performance loss due to premature prefetches and write-after-read stalls. In

contrast, these effects are more pronounced in RC since RC already overlaps all other write latencies.

Water is the only application helped unequivocally and significantly from the current optimizations to RC. It shows improvement in execution time from both prefetching (5.2% improvement over simple RC) and from speculative read execution (6.1% improvement over simple RC). The benefits are achieved by overlapping the prefetch of the critical section lock for the force updates, and the data within the critical section. Since the critical sections have low contention, the prefetch brings in valid data and is useful in hiding a large part of the latency. However, benefits from these techniques are far less significant than the corresponding benefits in SC. Erlebacher, FFT, and Mp3d see performance slowdowns with the addition of hardware prefetching and speculative read execution. The performance slowdown in these applications is mainly caused due to the negative effects of write prefetching like write-after-read stalls and increased false sharing due to early prefetches.

4.2.2 Impact of Write-Back L1 Caches

Overall, since RC already hides most of the write latency, the choice of the first-level cache does not have significant impact on the performance of RC on all our applications except Radix. The impact of the various optimizations with first-level write-back caches is qualitatively similar to that with write-through caches.

For a given RC configuration, Radix sees a significant reduction in execution time by replacing write-through caches with write-back caches. The difference arises with Radix only because it has a bursty irregular write pattern which overwhelms the secondary cache. Eventually, writes to remote data fill up the MSHRs, causing backup of subsequent requests, including write-throughs from L1. This resource backup eventually reaches into the L1 and the processor memory unit, adding contention to both reads and memory writes. With an L1 write-back cache, many writes hit in the L1; since these writes do not propagate to the L2, they relieve some of the contention

and saturation present in the write-through configuration. Erle, Mp3d and LU show a marginal degradation in performance when replacing a write-through cache with a write-back cache. The reason is that our write-through cache is a no-write-allocate cache while the write-back cache is a write-allocate cache; bringing the writes into the first-level write-back cache exacerbate conflicts within the cache.

4.2.3 Summary for RC

For our applications, the optimizations used in RC do not provide much benefit; the best improvement in execution time was 6.1% for Water. For four applications, the optimizations did not make a difference or resulted in a performance degradation. Thus, our experiments indicate that for RC, the cost of the on-chip hardware for the optimizations may not be justified. Regarding L1 cache write policy, our results show that except for one application, write-through L1 caches performed comparable to write-back L1 caches for RC.

4.3 Comparing SC and RC

Our results so far show that for our applications, the simplest RC implementation outperforms the most optimized SC. This is especially pronounced in the case with write-through L1 caches, where simple RC provides over a factor of two speedup for FFT and LU, a speedup of 1.5 or more for Erlebacher, Radix and Mp3d. (In Water, SC gets to within 5% of RC.) The performance difference between SC and RC is primarily caused by write latencies in SC that hardware prefetching and speculative reads are unable to hide for reasons discussed in Section 4.1.

In the case of write-back L1 caches, the performance improvement of RC over SC is less dramatic, but still significant. Two applications (Radix and Mp3d) see a speedup of 1.5 or more, and two other applications (LU and FFT) see a speedup of more than 1.25. Water and Erle show comparable performance with SC and RC (RC has speedups of 1.07 and 1.13 respectively). The difference in the results for write-

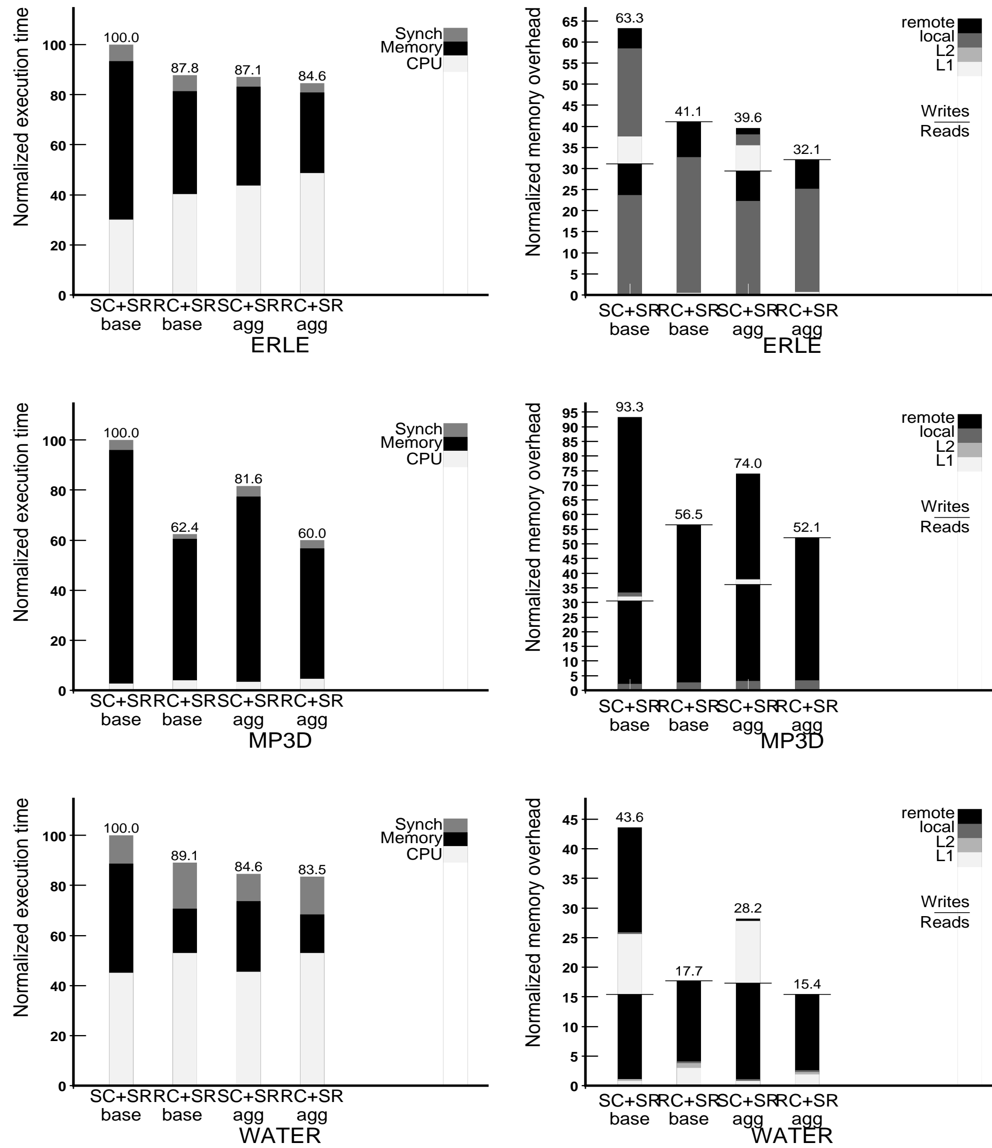
through and write-back L1 caches arises because SC variants improve in absolute cycle count from write-through to write-back, while the absolute cycle counts seen by RC systems stay the same in both configurations, except for Radix.

Thus, while SC needs a write-back cache for best performance, the performance of RC is largely independent of cache write policy. Accounting for possible additional latencies of having a write-back cache, the gap between RC (which can run just as well on a write-through L1 cache) and a high-performance SC (which needs a write-back L1 cache for best results) may increase further. Finally, either write-through or write-back primary caches with multiple cycle latencies are also likely to increase the gap between SC and RC.

4.4 Impact of a More Aggressive Protocol

As discussed in Section 3.1.1, to model a protocol with reasonable complexity, our caches delay ownership requests for writes to lines with pending shared reads. Thus, a write-prefetch seeking to obtain ownership of a cache line would be delayed if the cache had an earlier read miss to the same line. Although this implementation is representative of many current systems, a more aggressive system could allow such ownership requests to be sent on to the directory in parallel with the outstanding demand read access. However, the system would now need to handle possible races from network reordering at the directory and cache; this aggressive system may improve the overlap of ownership requests, but at the cost of added design complexity. Such an enhancement may have an impact on applications with migratory read-write sharing such as Water and MP3D, or on applications with producer-consumer sharing such as Erlebacher and LU where the producer reads the old value of the data before producing a new value.[||] However, this protocol enhancement will not have much impact

[||] In Erlebacher, some writes following reads to the same word access non-boundary planes in the 3-D array; such planes are accessed by only one processor. For such accesses, a four state protocol with a valid-exclusive state can get benefits similar to a protocol that overlaps read and ownership requests to the same line.



(a) Execution time components (b) Memory time components

Figure 4.6 Effect of aggressive coherence protocol

on applications such as Radix or FFT, since we observe that these two applications do not see delayed ownership requests in the key sections of their code. LU exhibits a high L2 write hit rate of more than 95%. Consequently, on our architecture, delayed ownership requests are not likely to have a significant effect on performance. In this section, we therefore examine the three applications, Erlebacher, Water, and Mp3d, where an aggressive coherence protocol is likely to impact performance to study the impact of a more aggressive coherence protocol on the performance of optimized implementations of consistency models on ILP multiprocessors.

We did not simulate the above aggressive protocol because of its significantly higher complexity. Instead, to approximate the impact of such a system, we inserted (by hand) explicit software exclusive prefetch instructions immediately before all read operations that are soon followed by a write to the same word, for Water, MP3D, and Erlebacher. Figure 4.6 summarizes our results comparing the performance of the three applications on the base system and a system with an aggressive coherence protocol, represented by `agg`, for the versions of SC and RC that support speculative read execution and hardware write prefetching (`SC+SR` and `RC+SR` respectively). Figure 4.6(b) shows the various components of the memory stall time in each system. We focus only on the system with a first-level write-back cache since the results in Section 4.3 show that write-back caches significantly outperform write-through caches for the SC implementations.

For SC, with the aggressive protocol, all three applications see more than 10% improvements in execution time due to reduced write latency and reduced write-after-read stalls (13% with Erlebacher, 15% with Water, and 18% with Erlebacher). For RC, all three applications see smaller benefits (less than 6%) with the aggressive coherence protocol since RC already hides write latencies effectively. The improvements in Water are due to more effective write prefetching which leads to faster releases and consequently faster acquires.

Comparing SC and RC, we find that using an aggressive coherence protocol has narrowed the performance gap between SC and RC by reducing SC's write limitations on all the three applications. Nevertheless, in MP3D, even with the aggressive coherence protocol, an RC implementation using the base coherence protocol still sees a reduction in execution time of 27% compared to the optimized implementation of SC. Erlebacher shows nearly equal performance for the most optimized SC and simple RC systems. Water stands apart from the other two applications; the best SC actually performs *better* than the base implementation of RC. Even though the RC also sees benefits from the more aggressive protocol for Water, the best RC still performs comparable to SC (within 1%).

Overall, these results suggest that the gap between SC and RC performance can be decreased by adding complexity to the cache-coherence protocol for some applications; however, a significant gap still remains for several applications (in applications where this optimization is not applicable and in applications where this optimization is not sufficient to equalize the performance of SC and RC). Furthermore, RC does not need the additional support to achieve its level of performance.

Note that the results in this section should not be interpreted as indicative of the effects of software-controlled prefetching on the performance difference between SC and RC. We insert software exclusive prefetches *immediately before* all read operations; our results therefore do not show the latency-hiding benefits associated with software prefetching in the previous section, but instead only approximate the effects of removing write-after-read stalls with an aggressive cache coherence protocol. The interaction of software prefetching and consistency models on ILP multiprocessors is discussed elsewhere [32].

4.5 Summary

This chapter presented the result of our study evaluating the effects of hardware-controlled read and write prefetching, and the effects of speculative reads with

hardware-controlled write prefetching. We found that for SC, these two techniques enhanced performance considerably (giving a speedup of over a factor of 2 in some cases). For RC, however, the optimizations showed an execution time improvement greater than 5% for only one case. The simplest RC implementation significantly outperforms the most optimized SC. The difference in performance between the two models, however, depends on the write policy of the primary cache and on the complexity of the cache-coherence protocol. Four our fairly aggressive base cache-coherence protocol, the simplest RC implementation significantly outperforms the most optimized SC.

The difference is higher with write-through primary caches than with write-back primary caches, but remains significant for several applications in both cases. With a more complex cache-coherence protocol, SC achieves performance comparable to RC for two applications, but a significant performance gap remains for others. The performance of SC is highly sensitive to cache write policy and the aggressiveness of the cache-coherence protocol, while the performance of RC is generally stable across all implementations. Overall, our results show that RC hardware has significant performance benefits over SC hardware, and at the same time requires less system complexity with ILP processors.

Chapter 5

Related Work

5.1 Consistency Models

A number of consistency models have been proposed in the literature or have been implemented on real systems [1]. In this thesis, we consider two consistency models, sequential consistency (SC) [23] and release consistency (RC) [14], to represent the tradeoff between programmability and performance. SC represents the most intuitive consistency model while previous studies have shown that RC gives the best performance for hardware cache-coherent multiprocessor systems.

5.2 Evaluation of Consistency Models

Several studies have evaluated the performance of memory consistency models [11, 13, 16, 44]. This thesis presents the first execution-driven simulation study for consistency models for aggressive ILP processors, evaluating two performance-enhancing techniques for consistency models used in such processors. Two previous quantitative evaluations of memory consistency models have used relatively aggressive processors. Gharachorloo et al. studied simple implementations of SC and RC; further, their study was trace-driven (as opposed to execution-driven) and did not accurately model the effects of synchronization and network contention [13]. Zucker and Baer studied SC and RC, implementing SC both in a straightforward fashion and also with the prefetching optimization; however, the processors they inspected were single-issue and statically scheduled [44].

5.3 Aggressive Implementations of Sequential Consistency

A number of hardware optimizations have been proposed in the literature to enhance the performance of implementations of sequential consistency. Gharachorloo et al. proposed hardware-controlled non-binding prefetching and speculative read execution to improve the performance of sequential consistency [12]. Adve and Hill [3] proposed an implementation of sequential consistency that can alleviate some of the latency of writes by allowing certain operations that follow the write to be serviced as soon as the write is serialized (as opposed to waiting for all invalidations to be acknowledged). Other studies have proposed implementations of SC that use the ordering guarantees of restrictive network topologies to reduce the write latency seen in the system [24]. Gharachorloo et al. discuss a technique called store buffering to improve the performance of SC by buffering writes separately and stalling the system only on the next read operation that follows the write operation [11]. In this thesis, we study the performance of two techniques, hardware prefetching and speculative loads, that have been integrated into commodity microprocessors to improve the performance of sequential consistency. We do not evaluate the other techniques to improve the performance of sequential consistency since they are either too restrictive, require more aggressive, complex system implementations, or do not appear to give significant performance benefits.

5.4 Prefetching

There has been substantial work in prefetching for multiprocessors (e.g., [6, 27]). This paper evaluates the use of non-binding hardware-controlled prefetching that exploits an aggressive processor's existing instruction window, to enhance the performance of consistency models. The interaction of software-controlled prefetching with consistency models has been studied for simpler processors with blocking reads [16, 11]. There has only been one study on the interaction of software prefetching with con-

sistency models on ILP multiprocessors with non-blocking reads [32]. This study showed that while software prefetching was successful in reducing the gap between the performance of memory consistency models, a large gap still remained between the simplest RC and most optimized SC.

Chapter 6

Conclusions

The memory consistency model of a shared-memory multiprocessor plays a key role in determining system potential for tolerating latency as it specifies the extent to which the system can appear to overlap or reorder memory operations. Previous studies with shared-memory multiprocessors have shown that the release consistency model (RC) significantly outperforms sequential consistency (SC), though at the cost of increased programming complexity. However, most of those studies assumed statically scheduled processors with blocking reads, and all assumed simple implementations of the consistency models. Current and next generation microprocessors aggressively exploit instruction-level parallelism (ILP) using methods such as multiple issue, dynamic scheduling, and non-blocking reads. For such processors, two optimizations have been proposed to enhance the performance of consistency models. Qualitatively, these optimizations may seem to bring the performance of SC closer to RC, potentially making SC more attractive to build in hardware for its easier programmability.

6.1 Thesis Summary

This thesis provides the first quantitative evaluation of various implementations of SC and RC for aggressive ILP processors. We evaluated the effects of hardware-controlled read and write prefetching, and the effects of speculative reads with hardware-controlled write prefetching. We found that for SC, these two techniques enhanced performance considerably (giving a speedup of over a factor of 2 in some cases). For RC, however, the optimizations showed an execution time improvement greater than 5% for only one case. The difference in performance between the two models, how-

ever, depends on the write policy of the primary cache and on the complexity of the cache-coherence protocol. For our fairly aggressive base cache-coherence protocol, the simplest RC implementation significantly outperforms the most optimized SC. The difference is higher with write-through primary caches than with write-back primary caches, but remains significant in both cases (four applications show a speedup of 1.25 or more with RC for both cache configurations). With a more complex cache-coherence protocol, SC achieves performance comparable to RC for two applications, but a significant performance gap remains for others. The performance of SC is highly sensitive to cache write policy and the aggressiveness of the cache-coherence protocol, while the performance of RC is generally stable across all implementations. Overall, our results show that RC hardware has significant performance benefits over SC hardware, and at the same time requires less system complexity with ILP processors. Write latencies that hardware prefetching and speculative reads do not effectively hide remain the most important reason for the performance gap between SC and RC.

6.2 Future Work

Compiler optimizations. In choosing a consistency model, the hardware designer must consider both system performance and programmability. The techniques of this paper address hardware performance. However, though there has been some work on compiler optimizations with sequential consistency [36, 20, 21], SC at the application programming level also restricts compiler optimizations. To avoid these restrictions, it is likely that high-performance compilers will expose a release-consistent model to the applications programmer. If compilers mandate RC, then the improved performance and lower complexity of RC further favor supporting RC in hardware for systems with ILP processors. An interesting area of future research is to quantify the performance improvements due to compiler optimizations allowed by relaxed memory consistency models.

Other hardware optimizations. This study focused on currently used hardware techniques for enhancing the performance of consistency models. Our results show that the main difference between the performance of SC and RC is the write latencies seen in SC. We are currently investigating two hardware techniques that target this additional write latency in the SC system – write buffering and speculative graduation.

Write buffering [11] allows writes to graduate from the instruction window, blocking only on the first read to reach the head of the instruction window while writes are pending. Write buffering can relax the restrictions on the commit rate of writes and increase the write overlap in current optimized implementations of SC.

Speculative retirement [33] is an optimization that targets the limitation due to the store-to-load constraint of SC. Loads stalled only for previous incomplete stores are allowed to speculatively commit their values into the processor's architectural state and retire from the instruction window, freeing up space for later instructions. This technique can potentially eliminate the impact of the store-to-load constraint (potentially equalizing the performance of SC to that of more relaxed models), but at an additional cost in hardware.

Finally, *larger instruction windows* can also be used to further increase the performance of SC. Larger instruction windows can potentially increase the overlap available to reads and writes in the system. Unlike RC which benefits mainly from the increased read overlap, SC can benefit from increases in both the read and the write overlap. Larger instruction windows thus offer another hardware technique to address the store latency responsible for the performance difference between SC and relaxed models.

Bibliography

[1] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. In *IEEE Computer*, pages 66–76, December 1996. Also available as Rice University ECE Technical Report 9512 and Western Research Laboratory Research Report 95/7.

[2] S. V. Adve and M. D. Hill. A Unified Formalization of Four Shared-Memory Models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[3] S. V. Adve and M. D. Hill. Weak Ordering - A New Definition. In *Proceedings 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[4] V. S. Adve, J.-C. Wang, J. Mellor-Crummey, D. Reed, M. Anderson, and K. Kennedy. An Integrated Compilation and Performance Analysis Environment for Data Parallel Programs. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995.

[5] R. G. Covington, S. Dwarkadas, J. R. Jump, S. Madala, and J. B. Sinclair. The Efficient Simulation of Parallel Computer Systems. *International Journal of Computer Simulation*, 1:31–58, January 1991.

[6] F. Dahlgren and P. Stenstrom. Effectiveness of Hardware-Based Stride and Sequential Prefetching in Shared-Memory Multiprocessors. In *Proceedings of the 1st International Symposium on High Performance Computer Architecture*, 1995.

[7] Digital Equipment Corporation. *Alpha Architecture Reference Manual*, 1992.

[8] M. Dubois, C. Scheurich, and F. A. Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings 13th Annual International Symposium on Computer Architecture*, pages 434–442, Tokyo, Japan, June 1986.

[9] M. Dubois, J. C. Wang, L. A. Barroso, K. Lee, and Y.-S. Chen. Delayed Consistency and Its Effects on the Miss Rate of Parallel Programs. pages 197–206, November 1991.

[10] J. H. Edmondson, P. I. Rubinfeld, P. J. Bannon, B. J. Benschneider, D. Bernstein, R. W. Castelino, E. M. Cooper, D. E. Dever, D. R. Donchin, T. C. Fischer, A. K. Jain, S. Mehta, J. E. Meyer, R. P. Preston, V. Rajagopalan, C. Somanathan, S. A. Taylor, and G. M. Wolrich. Internal Organization of the Alpha 21164, a 300-MHz 64-bit Quad-issue CMOS RISC Microprocessor. *Digital Technical Journal*, 7(1):119–132, 1995.

[11] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors. In *Proceedings of Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–257, 1991.

[12] K. Gharachorloo, A. Gupta, and J. Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proceedings of the International Conference on Parallel Processing*, pages I355–I364, 1991.

[13] K. Gharachorloo, A. Gupta, and J. Hennessy. Hiding Memory Latency Using Dynamic Scheduling in Shared-Memory Multiprocessors. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 22–33, 1992.

[14] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 15–26, May 1990.

[15] J. R. Goodman. Cache consistency and sequential consistency. Technical Report Technical Report #61, SCI Committee, March 1989. Also available as Computer Sciences Technical Report #1006, University of Wisconsin, Madison, February 1991.

[16] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W.-D. Weber. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 254–263, May 1991.

[17] D. Hunt. Advanced Features of the 64-bit PA-8000. Hewlett Packard Company.

[18] Intel Corporation. *Pentium (r) Pro Family Developer's Manual.*

[19] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, 1992.

[20] A. Krishnamurthy and K. Yelick. Optimizing Parallel Programs SPMD Programs. In *Languages and Compilers for Parallel Computing*, pages 331–345, August 1994.

[21] A. Krishnamurthy and K. Yelick. Optimizing Parallel Programs with Explicit Synchronization. In *Proceedings SIGPLAN Conference on Programming Language Design and Implementation*, July 1995.

[22] D. Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of the 8th International Symposium on Computer Architecture*, pages 81–87, May 1981.

[23] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[24] A. Landin, E. Hagersten, and S. Haridi. Race-Free Interconnection Networks and Multiprocessor Consistency. In *Proceedings 18th Annual International Symposium on Computer Architecture*, pages 106–115, May 1991.

[25] MIPS Technologies, Inc. *R10000 Microprocessor User's Manual, Version 2.0*, December 1996.

[26] T. Mowry. *Tolerating Latency through Software-controlled Data Prefetching.* PhD thesis, Stanford University, 1994.

[27] T. Mowry and A. Gupta. Tolerating Latency Through Software-Controlled Prefetching. *Journal on Parallel and Distributed Computing*, pages 87–106, June 1991.

[28] V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors. In *Proceedings of the Third Workshop on Computer Architecture Education*, February 1997.

[29] V. S. Pai, P. Ranganathan, and S. V. Adve. The Impact of Instruction Level Parallelism on Multiprocessor Performance and Simulation Methodology. In *Proceedings of the 3rd International Symposium on High Performance Computer Architecture*, pages 72–83, February 1997.

[30] V. S. Pai, P. Ranganathan, S. V. Adve, and T. Harton. An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–23, October 1996.

[31] U. Rajagopalan. The Effects of Interconnection Networks on the Performance of Shared-Memory Multiprocessors. Master's thesis, Department of Electrical and Computer Engineering, Rice University, January 1995.

[32] P. Ranganathan, V. S. Pai, H. Abdel-Shafi, and S. V. Adve. The Interaction of Software Prefetching with ILP Processors in Shared-Memory Systems. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[33] P. Ranganathan, V. S. Pai, and S. V. Adve. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1997.

[34] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. The Impact of Architectural Trends on Operating System Performance. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 285–298, December 1995.

[35] C. Scheurich and M. Dubois. Correct Memory Operation of Cache-Based Multiprocessors. In *Proceedings 14th Annual International Symposium on Computer Architecture*, pages 234–243, Pittsburgh, PA, June 1987.

[36] D. Shasha and M. Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[37] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[38] J. E. Smith and A. R. Pleszkun. Implementation of precise interrupts in pipelined processors. In *Proceedings of the International Symposium on Computer Architecture*, 1985.

[39] Sparc International. *The SPARC Architecture Manual*, 1993. Version 9.

[40] Sun Microsystems. *The UltraSPARC Processor – Technology White Paper*, 1995.

[41] Sun Microsystems Inc. *The SPARC Architecture Manual*, January 1991. No. 800-199-12, Version 8.

[42] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.

[43] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *ACM Computer Architecture News.*, 23(4), Septemeber 1995.

[44] R. N. Zucker and J.-L. Baer. A Performance Study of Memory Consistency Models. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 2–12, 1992.