Recent Advances in Memory Consistency Models for Hardware Shared-Memory Systems

Sarita V. Adve, Vijay S. Pai, and Parthasarathy Ranganathan
Department of Electrical and Computer Engineering
Rice University
Houston, Texas
rsim@ece.rice.edu

Abstract—
The memory consistency model of a shared-memory system determines the order in which memory operations will appear to execute to the programmer. The memory consistency model for a system typically involves a tradeoff between performance and programmability. This paper provides an overview of recent advances in hardware optimizations, compiler optimizations, and programming environments relevant to memory consistency models of hardware distributed shared-memory systems.

We discuss recent hardware and compiler optimizations that exploit the observation that it is sufficient to only *appear* as if the ordering rules of the consistency model are obeyed. These optimizations substantially improve the performance of the strictest consistency model, making it more attractive for its programmability. Recent concurrent programming languages and environments, on the other hand, support more relaxed consistency models. We discuss several such environments, including POSIX threads, Java, and OpenMP.

I. Introduction

The memory consistency model of a shared-memory system specifies the order in which memory operations will appear to execute to the programmer. The memory consistency model affects the process of writing parallel programs and forms an integral part of the entire system design including the architecture, the compiler, and the programming language.

Figure 1 shows a program fragment to illustrate how the memory consistency model affects the programmer. The figure shows processor P1 producing some data (`Data1` and `Data2`) that needs to be consumed by processor P2. The variable `Flag` is used for synchronization. Processor P1 writes the value 1 into `Flag` to indicate that the data is produced; processor P2 repeatedly reads `Flag` until it returns the value 1 indicating the data is ready to be consumed. The program is written with the expectation that processor P2's reads of the data variables will return the new values written by processor P1. However, in many current systems, processor P2's reads may return the old values of the data variables, giving an unexpected result. The memory consistency model of a shared-memory system is a formal specification that determines what values the programmer can expect a read to return.

Uniprocessors provide a simple memory consistency model to the programmer that ensures that memory operations will appear to execute one at a time, and in the order specified by the program (or program order). Thus, a read in a uniprocessor returns the value of the *last* write to the same location, where *last* is uniquely defined by the program order. Uniprocessor hardware and compilers, however, do not necessarily execute memory operations one at a time in program order. They may reorder and overlap memory operations; as long as data and control dependences are maintained, the system *appears* to obey the expected memory consistency model.

In a multiprocessor, the notion of a *last* write is not as well-defined, and a more formal specification of the memory consistency model is required. The most intuitive model for multiprocessors is a natural extension of the uniprocessor model. This model, sequential consistency, requires that memory operations appear to execute one at a time in some sequential order, and the operations of each processor occur in program order [1]. A read in a sequentially consistent multiprocessor must return the value of the last write to the same location in the above sequential order.

Unlike uniprocessors, simply maintaining per-processor data and control dependences is insufficient to maintain sequential consistency in a multiprocessor. For example, in Figure 1, there are no data or control dependences among the memory operations of processor P1. Thus, a uniprocessor could allow P1's write to `Flag` to be executed before its writes to the data variables. In this case, it is possible that P2 executes all its reads before the data writes of P1, returning the old values for the data variables and violating sequential consistency. This simple example shows that sequential consistency imposes more restrictions than simply preserving data and control dependences at each processor.

Sequential consistency can restrict several common hardware and compiler optimizations used in uniprocessors [2]. For this reason, relaxed consistency models have been proposed that explicitly permit relaxations of some program orderings. These models allow more optimizations than sequential consistency, but present a more complex model to the programmer. Thus, choosing the memory consistency model of a multiprocessor system has typically involved a tradeoff between performance and programmability. A tu-

This work is supported in part by IBM, Intel, the National Science Foundation under Grant No. CCR-9410457, CCR-9502500, CDA-9502791, and CDA-9617383, and the Texas Advanced Technology Program under Grant No. 003604-025. Sarita V. Adve is also supported by an Alfred P. Sloan Research Fellowship, Vijay S. Pai by a Fannie and John Hertz Foundation Fellowship, and Parthasarathy Ranganathan by a Lodieska Stockbridge Vaughan Fellowship.

Initially `Data1` = `Data2` = `Flag` = 0

P1	P2
`Data1` = 64	while (`Flag` != 1) {;}
`Data2` = 55	register1 = `Data1`
`Flag` = 1	register2 = `Data2`

Fig. 1. Motivation for a memory consistency model.

torial paper by Adve and Gharachorloo gives a detailed description of the optimizations restricted by straightforward implementations of sequential consistency as well as relaxations permitted by several relaxed models [2].

This paper provides an overview of recent advances in hardware optimizations, compiler optimizations, and programming languages and environments relevant to memory consistency models of hardware shared-memory systems. The advances in hardware and compiler optimizations seek to narrow the performance gap between memory consistency models, making stricter models more attractive for their programmability. On the other hand, recent concurrent programming languages and environments (e.g., POSIX threads, Java, and OpenMP) support more relaxed consistency models. We conclude with a discussion of the impact of the above advances.

II. Background

A. Hardware-centric Models

We briefly describe key aspects of the memory consistency models most commonly supported in commercial systems: sequential consistency [1] (supported by HP PA-8000 and MIPS R10000 processors), processor consistency [3] (similar to Sun's total store ordering and the model supported by Intel processors), weak ordering [4], and release consistency [3] (similar to Sun's relaxed memory ordering and the models supported by Digital Alpha and IBM PowerPC processors). More detailed descriptions of these models and their features can be found in [2].

The relaxed models mentioned above have been referred to as *hardware-centric* because they were primarily motivated by hardware optimizations. In this paper, optimizations regarding reordering a pair of memory operations implicitly refer to operations on different locations.

Figure 2 summarizes the relaxations in program order allowed by the various memory consistency models discussed in this paper. Sequential consistency requires program order to be maintained among all operations. Processor consistency allows reordering between a write followed by a read in program order. Weak ordering requires distinguishing between data and synchronization operations. Data operations can be reordered with respect to each other; however, program ordering of all operations with respect to synchronization operations must be maintained. Release consistency further categorizes synchronization operations into acquires and releases. Release consistency does not enforce ordering between two data operations, between a data operation and a subsequent acquire, or between a release and a subsequent data operation. In this paper, we use the term *release consistency* to refer to the RCpc model [3], which also does not enforce ordering between a release and a subsequent acquire. The distinctions between various categories of memory operations can be achieved in several ways [2]. Current processors support a conservative method through fence or memory barrier instructions. Typically, all program orders with respect to a fence or memory barrier instruction are enforced.

Model	Program Order Relaxation
Sequential consistency	None
Processor consistency	Write → Read
Weak Ordering	Data → Data
Release Consistency	Data → Data, Data → Acquire, Release → Data, Release → Acquire

Fig. 2. Relaxations of program order allowed by different memory consistency models. (Only program order between memory operations to different locations is considered.)

The examples in Figure 3 further illustrate the differences among the models from the programmer's viewpoint [2]. Part (a) repeats the example in Figure 1. With sequential consistency, since all memory operations must appear to occur in program order, it follows that P2's reads of `Data1` and `Data2` do not occur until P1's writes to the data locations complete. Therefore, P2's reads of the data locations will return the newly written values (64 and 55). Processor consistency ensures P1's writes will occur in program order and P2's reads will occur in program order; therefore, P2's reads must again return the newly written values. Weak ordering and release consistency, however, do not impose any such restrictions. Therefore, with these models, P1's writes to `Data1` and `Data2` may occur after P1's write to `Flag` and after P2's reads to `Data1` and `Data2`. Consequently, P2's reads of the data locations may return the old values of 0, giving unexpected results. With a weakly ordered or release consistent system, P2's reads can be made to return the new values if the programmer explicitly identifies all accesses to `Flag` as synchronization operations.

Figure 3(b) illustrates a simplified version of Dekker's algorithm for implementing critical sections. Sequential consistency prohibits the reads of both `Flag1` and `Flag2` from returning 0 in the same execution. Thus, with sequential consistency, at most one processor will enter the critical section. With processor consistency, however, the reads may execute before the writes of their respective processors (e.g., if the write is sent to a write buffer and the read is allowed to bypass the buffer). Therefore, both the reads of `Flag1` and `Flag2` may return 0 and both processors may enter the critical section, an undesirable result. With a weakly ordered or release consistent implementation also, both reads may return 0. To prohibit both processors from returning the value 0, the programmer must identify all operations to `Flag1` and `Flag2` as synchronization operations with weak ordering and release consistency, and use read-modify-writes with processor consistency [2].

The choice of the consistency model may limit the use of hardware and compiler optimizations that improve performance by overlapping, reordering, or eliminating memory operations (e.g., write buffers, lockup-free caches, out-of-order execution, register allocation, common sub-

Initially `Data1` = `Data2` = `Flag` = 0

P1	P2
`Data1` = 64	while (`Flag` != 1) {;}
`Data2` = 55	register1 = `Data1`
`Flag` = 1	register2 = `Data2`

Allowed results:

SC, *PC*: register1 = 64; register2 = 55

WO, *RC*: register1 = 64 or 0; register2 = 55 or 0

(a)

Initially `Flag1` = `Flag2` = 0

P1	P2
`Flag1` = 1	`Flag2` = 1
register1 = `Flag2`	register2 = `Flag1`
if (register1 == 0)	if (register2 == 0)
critical section	*critical section*

Allowed results:

SC: Both register1 and register2 cannot be 0

PC, *WO*, *RC*: register1 = 0 or 1; register2 = 0 or 1

(b)

Fig. 3. Illustration of memory consistency models. *SC* = sequential consistency, *PC* = processor consistency, *WO* = weak ordering, *RC* = release consistency

expression elimination, compile-time code motion, and loop transformations). In particular, reordering any pair of memory operations can potentially lead to a violation of sequential consistency. Weak ordering and release consistency allow the most optimizations since they allow reordering of all data operations between consecutive synchronization operations. However, these models are harder to reason with since they require programmers to deal with non-intuitive hardware and compiler optimizations. Thus, the choice of the memory consistency model of a system has typically involved a tradeoff between programmability and performance.

B. Programmer-centric Framework

The programmer-centric framework for specifying memory consistency models seeks to alleviate the tradeoff between programmability and performance seen by hardware-centric models [3], [5]. The framework is based on the hypothesis that programmers should not be required to reason about non-sequentially consistent systems. The framework guarantees sequential consistency if the programmer provides some information about the program or obeys certain constraints. The information and constraints relate only to the behavior of the program on sequentially consistent systems. The knowledge of this behavior is used by the system to determine optimizations that will not violate sequential consistency for that program.

The data-race-free [5] and properly-labeled [3] models illustrate the programmer-centric framework. These models seek to provide the optimizations of the weak ordering and release consistency hardware-centric models, but without requiring the programmer to reason about those optimizations. The models provide sequential consistency to data-race-free programs, which are defined as follows:

Call two memory operations *conflicting* if they are from different processors, they access the same location, and at least one of them is a write [6]. Assume the system provides a mechanism for distinguishing memory operations as data or synchronization.

A program is data-race-free if in every sequentially consistent execution of the program, any two conflicting memory operations are either separated by synchronization operations or are both distinguished as synchronization operations. Alternatively, if two conflicting operations occur one after another (without any intervening memory operations), then the programmer should distinguish them as synchronization operations (also called race operations); other operations may be distinguished as either data or synchronization.

For example, in Figure 3(a), the operations accessing `Data1` and `Data2` are always separated by the operations on `Flag` (in any sequentially consistent execution). Therefore, the operations on `Data1` and `Data2` may be distinguished as data or synchronization. However, the write of `Flag` and the final read of `Flag` are not separated by any other operations, and must be distinguished as synchronization.

The optimizations of weak ordering and release consistency can be applied to data-race-free programs without violating sequential consistency. The mechanism for distinguishing operations as data or synchronization may vary based on the programming language support (see Section V).

III. Hardware Advances

Straightforward hardware implementations of consistency models directly enforce their ordering constraints by prohibiting a memory operation O from entering the memory system until the completion of all previous operations for which O must appear to wait. For example, a straightforward implementation of sequential consistency will not issue a memory operation until the preceding memory operation of its processor is complete. However, certain features in recent processors, described below, permit more optimized implementations of consistency models.

Out-of-order scheduling. A processor with out-of-order scheduling simultaneously examines several consecutive instructions within an *instruction window*. The processor issues an instruction from the instruction window to its functional units or the memory hierarchy once the instruction's data dependences are resolved, even if previous instructions are still blocked. To maintain precise exceptions, however, instructions are retired from the instruction window and all changes to architectural state (e.g., registers and memory) are made in program order [7].

Non-blocking loads. Many current processors do not block on a load, but can continue executing independent instructions (including other loads) while one or more loads (including cache misses) are outstanding. To maintain precise interrupts, a load does not leave the instruction window until it returns a value. Therefore, the effectiveness of this technique is limited by the instruction window size, which determines the maximum number of instructions that will be overlapped with the load.

Speculation. Current processors support speculative execution in several forms, the most common of which is branch prediction. Instructions after the speculation point (e.g., a branch) continue to be decoded, issued, and executed as ordinary instructions. However, these instructions are not allowed to commit their values into the architectural state of the processor until all prior speculations have been resolved. If any speculation is determined to be incorrect (e.g., a mispredicted branch), the execution rolls back to the state at the point of the (mispredicted) speculation, and all later speculated instructions are squashed.

Prefetching. Many processors provide support for non-binding prefetch requests for cache lines that are likely to be accessed in the future. Such a request is expected to bring the accessed line in the processor's cache before the processor issues the actual demand access, thereby overlapping the memory latency with other work. Prefetch requests can either be initiated by software (through explicit prefetch instructions) or by hardware (using runtime prediction).

We next describe optimizations that use the above features to improve the performance of consistency models, and are supported in current commercial systems. These optimizations exploit the observation that it is sufficient to only *appear* as if the ordering rules of the consistency model are obeyed. Each of these techniques assumes a hardware cache-coherent system.

A. Hardware Prefetching from the Instruction Window

In straightforward implementations, a processor's instruction window may contain several decoded memory instructions that are not issued to the memory system due to consistency constraints. For example, a decoded load is not issued in a sequentially consistent system until the previous memory operation of the processor completes. The processor can issue non-binding prefetches for such instructions without violating the consistency model, thereby hiding some memory latency [8].

The latency that can be hidden using this technique is limited by the instruction window size. Additionally, prefetches that are issued too early may sometimes degrade performance by fetching data before it is ready. Such a degradation may result due to extra network traffic, and due to extra memory latency seen at processors that lose their cache lines prematurely to early prefetches.

B. Speculative Load Execution

The hardware prefetching technique described in the previous section does not allow a load to consume its value until the completion of preceding memory operations ordered by the consistency model, even if the load's value is already in the cache. Speculative load execution extends the benefits of prefetching by speculatively consuming the value of loads brought into the cache, regardless of consistency constraints [8]. If the accessed location does not change its value until the load could have been non-speculatively issued, then the speculation is successful. However, if the location does change its value, then the processor rolls back its execution until the incorrect load. The rollback can be achieved using the same mechanisms as used for flushing incorrectly executed instructions after a branch misspeculation or an exception.

To detect whether a value speculatively consumed by a load changes before the load is allowed to issue non-speculatively, the processor exploits the coherence mechanisms of hardware cache-coherent multiprocessors. In these systems, a change of a cache line by another processor will trigger an external coherence action (e.g., invalidate or update) to other caches holding the data, as long as the data remains in the cache. Thus, the processor monitors coherence requests to and replacements from its caches; either action to a cache line with speculatively consumed data triggers a rollback.

As with non-blocking loads in general, the latency tolerance potential of speculative load execution is limited by the size of the instruction window. In addition, consuming speculative values too early can result in increased rollbacks, potentially degrading performance.

Speculative load execution and hardware store prefetching from the instruction window are supported by the HP PA-8000, Intel Pentium Pro, and MIPS R10000 processors.

C. Cross-Window Prefetching

Prefetches may also be issued for instructions that are not currently in the instruction window, but are expected to be executed in the future. Either the compiler may insert explicit software prefetch instructions in the program or the hardware may issue such requests at runtime. Although this technique is applicable even in uniprocessor systems, use of cross-window prefetching to hide latency may also impact the relative performance of consistency models. This technique alleviates the limitations imposed by a small instruction window size for the hardware prefetching technique described in Section III-A, but requires the compiler or hardware to predict future accesses. Most current processors support a software prefetch instruction, and there has been significant compiler work for effective insertion of such prefetches (e.g., [9]). Several hardware techniques for predicting future data accesses have also been proposed (e.g., [10]), but are not yet commonly implemented.

D. Simulation Studies

There have been several simulation studies that analyze the performance differences between different memory consistency models. Earlier studies have analyzed straightforward implementations of consistency models [11], [12], and the optimization of hardware prefetching from the instruction window with single-issue, in-order processors [13]. More recent studies have examined the optimized implementations discussed in this section for multiprocessors built from state-of-the-art processors. These processors aggressively exploit instruction-level parallelism (ILP) with features such as multiple instruction-issue, out-of-order scheduling, non-blocking reads, and speculative execution [14], [15], [16]. This section summarizes results

ILP Processor	
Processor speed	300MHz
Maximum fetch/retire rate (instructions per cycle)	4
Instruction window	64 entries
Functional units	2 integer arithmetic 2 floating point 2 address generation
Memory queue size	32 entries
Cache parameters	
Cache line size	64 bytes
L1 cache (on-chip write-back)	Direct mapped, 16 K
L1 request ports	2
L1 hit time	1 cycle
L2 cache (off-chip write-back)	4-way associative, 64 K
L2 request ports	1
L2 hit time	8 cycles, pipelined
Number of MSHRs	8 per cache
Memory parameters	
Memory access time	18 cycles (60 ns)
Memory interleaving	4-way
Bus parameters	
Bus speed	100 MHz
Bus width	32 bytes
Network parameters	
Topology	2-dimensional mesh
Network speed	150MHz
Network width	64 bits
Flit delay (per hop)	2 network cycles

Fig. 4. Base system parameters.

Application	Input Size	Processors
FFT	65536 points	16
LU	256 by 256 matrix, block 8	8
MP3D	50000 particles	8
Radix	1024 radix, 512K keys, max 512K	8
Water	512 molecules	16

Fig. 5. Applications, input sizes, and system sizes

found in the latter studies; however, the quantitative data reported here are based on a new set of simulations with a uniform set of system parameters for all experiments, a more recent (and aggressive) compiler, and a more aggressive cache coherence protocol (MESI versus MSI).

Simulation framework.

We report results for five scientific applications from the Stanford SPLASH and SPLASH-2 suites [17], [18] on a simulated hardware cache-coherent shared-memory multiprocessor system. The simulations are performed with RSIM, the Rice Simulator for ILP Multiprocessors, a detailed execution-driven simulator [19]. RSIM models an out-of-order superscalar processor pipeline, a two-level cache hierarchy, a split-transaction bus on each processor node, and an aggressive memory and multiprocessor interconnection network subsystem, including contention at all resources. The modeled systems implement an invalidation-based four-state MESI directory cache coherence protocol. Figure 4 summarizes the values of the parameters for the base systems simulated. The cache sizes are chosen following the working set characterizations of Woo et al. [18]. We also performed experiments that double and quadruple all the miss latency components in the system; our results hold qualitatively even in this configuration.

The applications, their input sizes, and the number of processors in the system for each application are summarized in Figure 5. A 16-processor system is simulated for applications that scale well (FFT and Water), while an 8-processor system is simulated for LU, MP3D, and Radix.

Hardware prefetching from the instruction window and speculative load execution.

Figure 6 shows, for each application, the execution time for the straightforward implementations of sequential consistency (SC), processor consistency (PC), and release consistency (RC), normalized to the time for sequential consistency. (We do not consider weak ordering separately as our applications see very little synchronization overhead; therefore, we expect few differences between weak ordering and release consistency.) Figure 7 shows, for each application, the execution times for the best implementation of the three consistency models normalized to the time for the straightforward implementation of sequential consistency in Figure 6. Comparing the corresponding bars of Figures 6 and 7 shows the impact of the hardware optimizations.

The optimizations of hardware prefetching from the instruction window and speculative load execution provide significant benefits for the more constrained consistency models (sequential consistency and processor consistency), but do not significantly impact release consistency. The optimizations greatly narrow the performance gap between the various models for all applications. Nevertheless, processor consistency and release consistency continue to show sizeable performance benefits compared to sequential consistency for two of our five applications, in all of the configurations studied (release consistency shows 25% or more reduction in execution time for Radix and MP3D). The differences in performance between the various models seen in Figure 7 stem primarily from (1) limited hardware resources (mainly the instruction window), which limit the extent to which the optimizations can be exploited, and (2) the negative effects of early store prefetches, which lead to additional exposed latencies in the stricter models.

Increasing the instruction window size.

The instruction window size largely determines the effectiveness of the optimizations of hardware prefetching from the instruction window and speculative load execution. Increasing the size of the instruction window and memory queue generally narrows the remaining performance gap between memory consistency models. For example, doubling the instruction window and memory queue sizes reduces the difference in execution time between the best SC and RC versions to 16% and 19% for MP3D and Radix respectively (compared to 24% and 28% with the base configuration). However, in a few cases, larger increases in the instruction window and memory queue sizes lead to performance degradations in sequential consistency and processor consistency, widening the performance gap with release consistency. This degradation occurs because fundamental limitations of hardware prefetching and speculative loads (caused by negative effects of early prefetches and rollbacks with speculative loads) were exposed or exacerbated with larger instruction window sizes.

Cross-window software prefetching.

Software prefetches were inserted in the applications by hand [15], following a state-of-the-art prefetching algo-

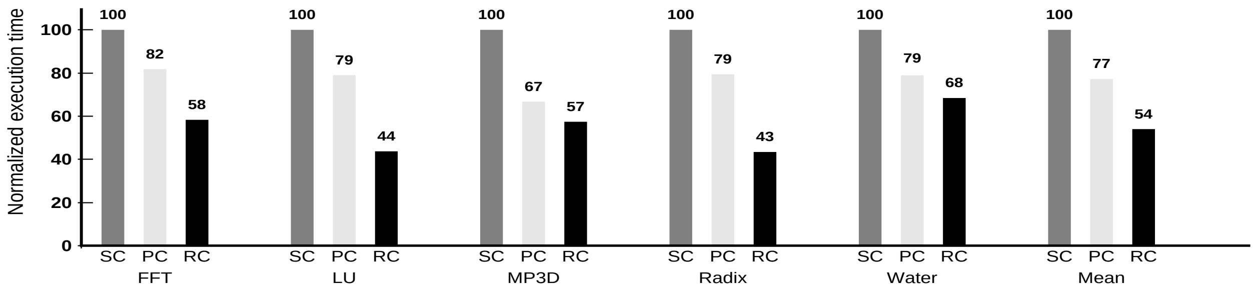

Fig. 6. Performance of straightforward implementations of memory consistency models

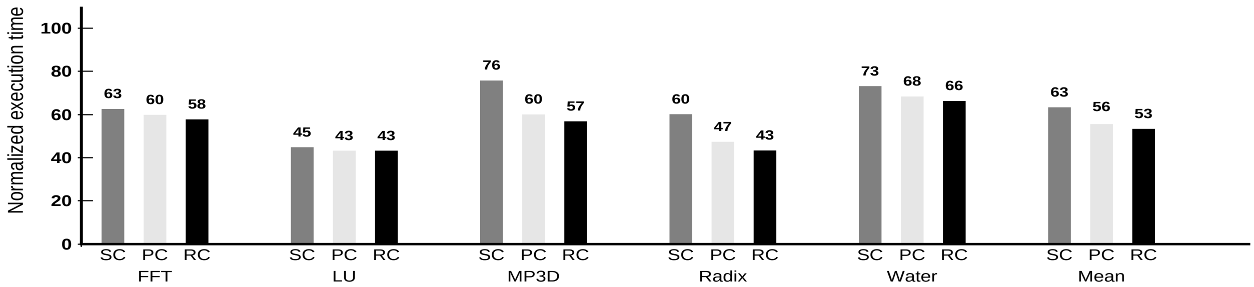

Fig. 7. Impact of hardware prefetching and speculative execution on consistency models (normalized to straightforward SC)

rithm [9]. Software prefetching improves the performance of all the consistency models (with and without the optimizations of hardware prefetching from the instruction window and speculative load execution) [15]. However, MP3D and Radix continue to see significant improvements with processor consistency and release consistency compared to sequential consistency (more than 20% reduction in execution time with release consistency).

Limitations of software prefetching on multiprocessors with aggressive processors (due to late or early prefetches, or resource contention) and memory latencies that are not amenable to prefetching (due to limitations of the software prefetching algorithm and contention at processor resources) are responsible for the remaining gap between the consistency models.

E. Other Optimizations

We have focused on optimizations implemented in current commercial systems at the processor level. Several other processor-centric optimizations have also been proposed and evaluated in the literature for hardware shared-memory systems. These include techniques to overlap both data latency (e.g., multithreading [20]) and synchronization latency (e.g., multithreading, and fuzzy and selective acquires [14]). The simulation studies described above found that data latency is far more dominant than synchronization latency for the applications and systems studied; therefore, future techniques that target data latency appear to be more likely to benefit performance for this class of systems and applications.

System design decisions below the processor level also influence the performance of memory consistency models. For example, the cache write policy and cache coherence protocol can impact the relative performance of consistency implementations [14]. These policies affect the overhead of writes, and thus generally have a greater influence on consistency models where write latencies are not already completely overlapped (SC and PC). RC is less sensitive to these choices. This paper uses the best practical configurations for all models (i.e., writeback caches and MESI protocol).

Finally, even if a processor supports a strict consistency model, the underlying system design decisions may result in a more relaxed model. For example, even if the processor supports sequential consistency, the memory controller may acknowledge a data store to the processor before it is actually globally visible. The net result is a system that is more aggressive than the consistency model supported by the processor, but somewhat more conservative than one in which the processor supports a relaxed consistency model. The Convex SPP 2000 system is an example of a commercial system that includes a weak ordering mode even though its processors (HP PA-8000) support sequential consistency. The Stanford FLASH system also uses a similar approach.

F. Summary

In summary, recent hardware optimizations result in a significant narrowing of the performance gap between consistency models, virtually eliminating the gap for three of the five applications studied on our base system. Nevertheless, processor consistency and release consistency show significant performance benefits over sequential consistency for two of our applications on all the system configurations we studied. Thus, for the class of applications and systems studied here, the choice of the consistency model for future systems will depend on the importance of the remaining hardware performance gap to system vendors, as well as the impact of relaxed consistency models on compiler optimizations and programming language support for relaxed

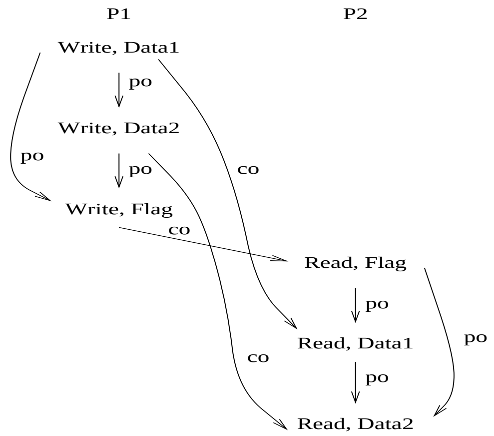

Fig. 8. Execution graph corresponding to Figure 1

consistency models. The rest of the paper discusses the latter two issues.

IV. Compiler Advances

Sequential consistency and processor consistency restrict the direct application of several uniprocessor compile-time optimizations that may involve reordering or eliminating memory operations (including register allocation). In their pioneering work, Shasha and Snir developed a compile-time analysis to identify memory operations that can be reordered without violating sequential consistency [6].

To understand the analysis by Shasha and Snir, consider an execution of a parallel program represented as a graph with the following properties. The vertices of the graph are the memory operations in the execution. There is an edge from operation A to operation B in the graph if either (i) A precedes B in program order, or (ii) A precedes B in the execution and A and B conflict.[1] Call the two categories of edges program order (*po*) and conflict order (*co*) edges respectively, and call the graph the execution graph. Figure 8 illustrates an execution graph for an execution of the program in Figure 1 in which the data reads by processor P2 return the new values written by processor P1.

Shasha and Snir observed that if the execution graph is acyclic, then the execution appears sequentially consistent (since all memory operations can be totally ordered in an order consistent with program order). For example, the execution graph in Figure 8 is acyclic and the depicted execution is sequentially consistent.

To ensure the execution graph is acyclic, it is sufficient to ensure that if there is a program order edge from A to B on a potential cycle in the graph, then A and B are not reordered. Operations on all other program order edges can be reordered without violating sequential consistency. For the program in Figure 8, potential cycles in the execution graph are:

- Write, Data1 $\xrightarrow{po}$ Write, Flag $\xrightarrow{co}$ Read, Flag $\xrightarrow{po}$ Read, Data1 $\xrightarrow{co}$ Write, Data1
- Write, Data2 $\xrightarrow{po}$ Write, Flag $\xrightarrow{co}$ Read, Flag $\xrightarrow{po}$ Read, Data2 $\xrightarrow{co}$ Write, Data2
- Write, Data1 $\xrightarrow{po}$ Write, Data2 $\xrightarrow{po}$ Write, Flag $\xrightarrow{co}$ Read, Flag $\xrightarrow{po}$ Read, Data1 $\xrightarrow{co}$ Write, Data1
- Write, Data2 $\xrightarrow{po}$ Write, Flag $\xrightarrow{co}$ Read, Flag $\xrightarrow{po}$ Read, Data1 $\xrightarrow{po}$ Read, Data2 $\xrightarrow{co}$ Write, Data2

With the analysis so far, all program order edges shown in Figure 8 could be on a potential cycle, prohibiting meaningful optimizations. Shasha and Snir further formalized a minimal set of cycles, called critical cycles, and showed that it is sufficient to consider only program order edges on critical cycles. Operations that are not ordered by such critical program order edges can be reordered without violating sequential consistency. Applying their formalization (not discussed here further due to lack of space), only the first two cycles mentioned above for Figure 8 are critical. Therefore, program ordering needs to be maintained only with respect to accesses to Flag; the accesses to the data locations can be reordered with respect to each other.

At compile time, it is difficult to predict or analyze all executions and their execution graphs. Therefore, Shasha and Snir apply their analysis to a graph where the vertices are the static memory operations in the program, and a bi-directional conflict order edge is introduced for every pair of static memory operations that could conflict in any execution.

More recently, Krishnamurthy and Yelick showed that Shasha and Snir's algorithm for detecting critical cycles has exponential complexity in the number of processors. They developed an algorithm for SPMD programs with polynomial complexity [21]. They also evaluated the effect of certain optimizations using their algorithm, on array-based Split-C programs run on a CM-5 (message passing) multiprocessor. They found reductions in execution time of 20% to 50%. They further improved their algorithm to reduce the impact of bi-directional conflict order edges [22], using some information from the programmer. This improvement reduced the number of critical cycles for which program order needs to be enforced, and gave additional reductions in execution time of 20% to 35% for Split-C programs on a CM-5 multiprocessor.

It is not yet known how effective the above analyses and optimizations are for general programs on native shared-memory machines, or how they compare to compile-time optimizations enabled by a relaxed consistency model such as release consistency.

V. Programming Languages

Until recently, most memory consistency models were developed primarily by computer architects for the hardware interface. Many common high-level programming languages did not include standard support for explicitly parallel shared-memory programs, and did not explicitly consider the issue of the memory consistency model. Programmers typically relied on parallelizing compilers, or used non-standard vendor specific extensions to generate shared-memory parallelism and synchronization. Recent languages and programming environments, however, deal with shared-memory parallelism and the issue of memory consistency models more explicitly. Below, we discuss com-

[1] Recall that two operations conflict if they are from different processors, they access the same location, and at least one is a write.

monly used languages and environments that are currently supported by multiple system vendors, and conclude with a discussion on the implications for the compiler.

A. POSIX Threads

POSIX is an IEEE standard [23] that includes a threads interface for the C language. It specifies several synchronization functions; e.g., functions to implement mutual exclusion locks and condition variables. For memory consistency, POSIX requires applications to use the provided synchronization functions to separate conflicting accesses to the same memory location. Such applications can rely on sequentially consistent results. Applications that synchronize in other ways (as in the examples in Figure 3) or that include races among user data structures (e.g., asynchronous algorithms) may get unexpected results.

The POSIX threads model is like the data-race-free/properly-labeled model described in Section II-B in that it requires synchronization to be explicit, to ensure reliable results. However, it does not provide the full flexibility of the data-race-free/properly labeled model since it restricts synchronization to only the provided synchronization functions. The provided synchronization functions may be overkill for certain cases (e.g., Figure 3(a)), potentially resulting in a loss of performance. The data-race-free/properly labeled model conceptually permits any memory operation (including races in asynchronous programs) to be distinguished as synchronization.

B. The `volatile` Declaration

The ANSI C, C++, and Java languages support the `volatile` declaration to suppress certain optimizations. A key motivation for this declaration was to inhibit optimizations in codes such as device drivers and interrupt handlers. The declaration is often also used for enforcing data consistency in shared-memory programs.

The Java language specification provides the most precise specification for the semantics of the `volatile` declaration [24]. Every access to a `volatile` variable must result in a memory access; i.e., values of such variables cannot be cached in registers and accesses to such variables cannot be eliminated through optimizations such as common sub-expression elimination. Furthermore, program-ordered accesses to `volatile` variables cannot be reordered with respect to each other. There is no restriction on the ordering between an access to a `volatile` variable and an access to a non-`volatile` variable. For example, for the program in Figure 3(a), the variables `Data1`, `Data2`, and `Flag` must all be declared `volatile` to ensure that the reads of `Data1` and `Data2` will return the new values written in the program. This, however, unnecessarily precludes optimizations on accesses to `Data1` and `Data2`.

The above example illustrates that the current specification is not restrictive enough to use `volatile` as the sole mechanism for enforcing consistency, while enabling common optimizations. An additional constraint of preserving program order between non-`volatile` and `volatile` accesses would have provided a model similar to weak ordering and release consistency. Java provides another mechanism to overcome this deficiency, as discussed next.

C. The Java Programming Language

This section discusses the high-level Java programming language, as opposed to the Java virtual machine. The formal memory consistency model for Java is defined as a set of rules on an abstract representation of the system [24]. The abstraction consists of a main memory (containing the master copy of all variables) and a working memory per thread. A thread's instructions result in `use` and/or `assign` operations on the values in its working memory. The `use` and `assign` operations (and the lock and unlock operations described below) occur in program order. The underlying implementation transfers data between the working and main memories, subject to several ordering rules. The current Java model is complex and hard to interpret; only the key ordering rules are informally described below.

A Java program must use the `volatile` or `synchronized` keywords to enforce memory ordering. `Volatile` is common to C and C++ and is discussed in Section V-B. The `synchronized` keyword results in a lock access on the associated object before executing the corresponding statement or method, and an unlock access afterward. The key ordering rules are: (i) A lock access must appear to flush each variable V from its thread's working memory before the thread's next use of V, unless the thread makes an assignment to V between the lock and the next use of V. (ii) Before an unlock access, all values previously assigned by that thread must be copied to main memory. Thus, a thread's load following a lock will either get a value assigned by that thread after the lock, or a value that is at least as recent as in the master copy at the time of the lock. After an unlock, all the values from the thread's preceding assignments will be in the master copy.

Referring to Figure 3(a), the `synchronized` keyword need be applied only to the accesses to `Flag` to ensure that the reads of the data locations will return the correct values. In contrast to the use of the `volatile` declaration discussed in Section V-B, the use of `synchronized` allows the writes (and reads) of `Data1` and `Data2` to be reordered with respect to each other without affecting the result of the execution. To prevent excessive spinning on locks, Java also provides `wait` and `notify` primitives for producer-consumer synchronization. Nevertheless, these primitives also require lock accesses for effective use, which appears to be overkill for interactions such as in Figure 3(a).

The ordering rules for Java are almost similar to those for release consistency.[2] Java is also accompanied by a pro-

[2]There are two subtle differences between Java and release consistency. First, in a Java thread, the presence of a lock between a write and a read to the same location requires that the write appear to be serialized at main memory before the read, unless there is another write to the same location between the lock and the read [24]. Release consistency does not require such a serialization. Second, unlike release consistency, Java does not allow program-ordered reads to the same location to be reordered [24]. As mentioned earlier, this paper does not discuss ordering constraints between memory operations to the same location.

gramming style recommendation similar to the data-race-free/properly-labeled model. It states that "if a variable is ever to be assigned by one thread and used or assigned by another, then all accesses to that variable should be enclosed in `synchronized` methods or `synchronized` statements."

D. OpenMP

OpenMP is a recently proposed application programming interface for portable shared-memory programs [25]. It provides relatively high-level directives to specify parallel tasks or loops, and various synchronization constructs (e.g., critical sections, atomic updates, and barriers). For memory consistency, it supports a `FLUSH` directive that must be used to ensure all preceding writes of a processor are seen by the memory system and subsequent reads return new values. A `FLUSH` is also implicitly associated with many of the synchronization constructs. The description of `FLUSH` in the current specification is fairly informal (e.g., it does not explicitly state the impact of a `FLUSH` on preceding reads or subsequent writes), but it appears to provide a model similar to weak ordering and release consistency and has semantics similar to memory barrier or fence instructions supported by most current processors. Thus, to obtain the expected result for the example in Figure 3(a), the application needs to simply include `FLUSH`es with the accesses to `Flag`. Additionally, OpenMP allows a `FLUSH` to be explicitly applied to only a specified list of variables.

E. Message Passing Interface (MPI)

The issue of the memory consistency model is generally not relevant to traditional message passing applications because synchronization is implicit with every data transfer message. The new Message Passing Interface (MPI-2) standard [26], however, also includes support for one-sided communication, which has similarities with shared-memory and must consider the consistency model. We briefly discuss relevant aspects of MPI-2 below.

The key primitives for one-sided communication are `get` and `put`. These allow a processor to load or store a data buffer from a remote processor's memory without any corresponding `send` or `receive` by the remote processor's program. Special synchronization mechanisms are provided to ensure that the loads return, and stores deposit, appropriate values at the appropriate time. Ordering rules analogous to release consistency are enforced; e.g., the user cannot rely on the completion of `get`s and `put`s until appropriate synchronization has occurred.

F. High Performance Fortran (HPF)

The High Performance Fortran (HPF) language [27] provides various constructs to expose data parallelism to the system, but provides sequential semantics to the programmer. Each HPF data parallel construct (including Fortran 90 array statements, the FORALL statement, and HPF intrinsics) has equivalent sequential semantics. Therefore, HPF programmers need not be concerned with the traditional notion of a shared-memory consistency model. One distinct aspect of the sequential model provided by HPF is that the data parallel array statements and the FORALL statement have copy-in/copy-out semantics, where all locations on the right hand side of an assignment are read before any locations on the left hand side are assigned. This provides straightforward deterministic sequential semantics even for a statement that may appear to have dependences among its various computations.

HPF also provides EXTRINSIC procedures to invoke other parallelism paradigms not directly supported in HPF (e.g., explicitly parallel code for branch-and-bound parallelism). Such a procedure may have an independent memory consistency model; the language requires all processors to appear to enter and exit such a procedure together.

G. Implications for the Compiler

The consistency model supported by the programming language influences the optimizations a compiler can perform. From the compiler's viewpoint, most recent programming environments support models similar to data-race-free/properly labeled, allowing significant flexibility [2]. For example, with POSIX, Java, and OpenMP, the compiler can reorder any pair of operations (to different locations) between two synchronization/`volatile`/`FLUSH` operations, as well as allocate memory in registers in this interval.

In addition, the compiler has the responsibility to ensure that the parallel program it generates runs correctly for the memory consistency model supported by the hardware. This is fairly straightforward for the languages and environments discussed in this section and for current commercial systems. For example, most current processors (and systems) support memory barriers or fences. For such systems, the special synchronization constructs in the above environments must include fences at all points in the construct where a memory operation may be involved in a race [2]. Typically, these fences would be inserted by library writers for the synchronization constructs, and the compiler simply must ensure that it maintains these fences and their orderings with respect to other memory operations in the final program. Similarly, the `FLUSH` directive of OpenMP must be replaced by a hardware fence. For HPF, the parallelizing compiler itself inserts synchronization, and must insert hardware fences at these points. A more comprehensive description of how to port data-race-free/properly labeled programs to common hardware memory consistency models appears in [28], [29].

VI. Concluding Remarks

This paper provides an overview of recent advances in memory consistency models, covering hardware, compilers, and programming environments. Recent hardware optimizations have significantly narrowed the hardware performance gap between various consistency models, virtually eliminating the gap for three of the five applications studied on our base system. Nevertheless, processor consistency and release consistency show significant benefits over sequential consistency for two of our applications on all the system configurations we studied.

Compiler optimizations relevant to consistency models have not been as extensively studied as hardware optimizations. Recent work has shown that it is possible to implement reordering optimizations in the compiler without violating sequential consistency. However, these optimizations have been evaluated for restricted programs and systems, and have not been compared with the benefits of relaxed models. The importance of relaxed models to compilers remains one of the key open questions in this area.

Recent programming languages and environments for explicitly parallel shared-memory programs (e.g., POSIX threads, Java, and OpenMP) support relaxed consistency models. For many programs, some of these languages (e.g., POSIX threads and Java) require that for expected results, system-provided synchronization constructs be used to prevent data races. While such an approach encourages good programming practice, the provided constructs may be overkill or inappropriate for some cases, leading to a performance loss compared to more flexible approaches. OpenMP provides more flexibility, allowing synchronization through arbitrary reads and writes; data consistency is achieved by using the `FLUSH` directive at the appropriate (synchronization) points in the program.

High-level languages with relaxed models eliminate the programmability advantage of supporting sequential consistency in the compiler or hardware, from the perspective of programmers of such languages. However, hardware and compiler designers must consider the programmability/performance tradeoff for programmers of other languages (either high-level or assembly level) before deciding the consistency model. This tradeoff will depend on results of future studies on compiler optimizations, and the importance of the remaining hardware performance gap to system vendors. Finally, the memory consistency model must be chosen to last beyond current implementations since it is an architectural specification visible to the programmer.

VII. Acknowledgments

We are grateful to Bill Pugh for correcting the description of the Java memory model in an earlier version of this paper. We thank Vikram Adve and Kourosh Gharachorloo for valuable comments on earlier versions of this paper. We also thank Vikram Adve and Rohit Chandra for clarifications on HPF and OpenMP respectively.

References

[1] L. Lamport, “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs,” *IEEE Trans. on Computers*, vol. C-28, no. 9, pp. 690–691, September 1979.

[2] S. V. Adve and K. Gharachorloo, “Shared Memory Consistency Models: A Tutorial,” *IEEE Computer*, pp. 66–76, Dec. 1996.

[3] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, “Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors,” in *Proc. 17th Intl. Symp. on Computer Architecture*, pp. 15–26, May 1990.

[4] M. Dubois, C. Scheurich, and F. A. Briggs, “Memory Access Buffering in Multiprocessors,” in *Proc. 13th Intl. Symp. on Computer Architecture*, pp. 434–442, June 1986.

[5] S. V. Adve and M. D. Hill, “Weak Ordering - A New Definition,” in *Proc. 17th Intl. Symp. on Computer Architecture*, pp. 2–14, May 1990.

[6] D. Shasha and M. Snir, “Efficient and Correct Execution of Parallel Programs that Share Memory,” *ACM Trans. on Programming Languages and Systems*, vol. 10, no. 2, pp. 282–312, April 1988.

[7] J. E. Smith and A. R. Pleszkun, “Implementation of Precise Interrupts in Pipelined Processors,” in *Proc. 12th Intl. Symp. on Computer Architecture*, 1985.

[8] K. Gharachorloo, A. Gupta, and J. Hennessy, “Two Techniques to Enhance the Performance of Memory Consistency Models,” in *Proc. Intl. Conf. on Parallel Processing*, pp. I355–I364, 1991.

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

[10] T.-F. Chen and J.-L. Baer, “Effective Hardware-Based Data Prefetching for High-Performance Processors,” *IEEE Transactions on Caches*, vol. 44, pp. 609–623, May 1995.

[11] K. Gharachorloo, A. Gupta, and J. Hennessy, “Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors,” in *Proc. 4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 245–257, 1991.

[12] K. Gharachorloo, A. Gupta, and J. Hennessy, “Hiding Memory Latency Using Dynamic Scheduling in Shared-Memory Multiprocessors,” in *Proc. 19th Intl. Symp. on Computer Architecture*, pp. 22–33, 1992.

[13] R. N. Zucker and J.-L. Baer, “A Performance Study of Memory Consistency Models,” in *19th Intl. Symp. on Computer Architecture*, pp. 2–12, 1992.

[14] 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 *Proc. 7th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 12–23, 1996.

[15] 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 *Proc. 24th Intl. Symp. on Computer Architecture*, pp. 144–156, 1997.

[16] 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 *Proc. 9th Symp. on Parallel Algorithms and Architectures*, pp. 199–210, 1997.

[17] J. P. Singh, W.-D. Weber, and A. Gupta, “SPLASH: Stanford Parallel Applications for Shared-Memory,” *Computer Architecture News*, vol. 20, pp. 5–44, March 1992.

[18] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, “The SPLASH-2 Programs: Characterization and Methodological Considerations,” in *Proc. 22nd Intl. Symp. on Computer Architecture*, pp. 24–36, June 1995.

[19] V. S. Pai, P. Ranganathan, and S. V. Adve, “RSIM: A Simulator for Shared-Memory Multiprocessor and Uniprocessor Systems that Exploit ILP,” in *Proc. 3rd Workshop on Computer Architecture Education*, 1997.

[20] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W.-D. Weber, “Comparative Evaluation of Latency Reducing and Tolerating Techniques,” in *Proc. 18th Intl. Symp. on Computer Architecture*, pp. 254–263, May 1991.

[21] A. Krishnamurthy and K. Yelick, “Optimizing Parallel SPMD Programs,” in *Proc. 7th Workshop on Languages and Compilers for Parallel Computing*, 1994.

[22] A. Krishnamurthy and K. Yelick, “Optimizing Parallel Programs with Explicit Synchronization,” in *Proc. SIGPLAN Conf. on Programming Language Design and Implementation*, July 1995.

[23] *Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API) [C Language]*. IEEE, July 1996. ISO/IEC standard 9945-1:1996(E), ANSI/IEEE Standard 10003.1.

[24] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, pp. 399–417. The Java Series, Addison-Wesley, 1996.

[25] *OpenMP Fortran Application Program Interface – Version 1.0*, October 1997.

[26] *MPI-2: Extensions to the Message-Passing Interface*. July 1997.

[27] C. Koelbel et al. *The High Performance Fortran Handbook*. The MIT Press, 1994.

[28] S. V. Adve, *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Computer Sciences Department, University of Wisconsin-Madison, December 1993.

[29] K. Gharachorloo, *Memory Consistency Models for Shared Memory Multiprocessors*. PhD thesis, Stanford University, 1995.