Multiprocessors Should Support Simple Memory Consistency Models

Mark D. Hill

Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton St.
Madison, WI 53706 USA
http://www.cs.wisc.edu/~markhill

Abstract

Many future computers will be shared-memory multiprocessors. These hardware systems must define for software the allowable behavior of memory. A reasonable model is sequential consistency (SC), which makes a shared memory multiprocessor behave like a multiprogrammed uniprocessor. Since SC appears to limit some of the optimizations useful for aggressive hardware implementations, researchers and practitioners have defined several relaxed consistency models. Some of these models just relax the ordering from writes to reads (processor consistency, IBM 370, Intel Pentium Pro, and Sun TSO), while others aggressively relax the order among all normal reads and writes (weak ordering, release consistency, DEC Alpha, IBM PowerPC, and Sun RMO).

This paper argues that multiprocessors should implement SC or, in some cases, a model that just relaxes the ordering from writes to reads. I argue against using aggressively relaxed models because, with the advent of speculative execution, these models do not give a sufficient performance boost to justify exposing their complexity to the authors of low-level software.

Keywords: multiprocessors, parallel computing, shared memory, memory consistency models.

1 Introduction

Many future computers will contain multiple processors, in part, because the marginal cost of adding a few additional processors is so low that only minimal performance gain is needed to make the additional processors cost-effective [11]. Intel, for example, now makes cards containing four Pentium Pro™ processors that can easily be incorporated into a system. Multiple-processor cards will help multiprocessing spread from servers to the desktop.

Personal use of this material is permitted. However, permission to reprint/ republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

(C) 1998 IEEE.

How will these multiprocessors be programmed? The evolution that has already begun is likely to continue. First, multiprocessors are used for multiprogramming, where conventional single-threaded programs are multiplexed on the processors. Next, performance-critical parts of compute-intensive applications will be parallelized by expert programmers to use multiple threads sharing data through shared memory. When one game vendor, for example, parallelizes and obtains a performance advantage, competitors will rapidly follow suit. Finally, someday we may be able to build compilers that can effectively parallelize most sequential programs or provide tools and abstractions that allow many people to program in parallel.

What hardware is needed to support threads with shared memory? First, the hardware should provide a well-defined *interface* for shared memory. Second, it should provide a high-performance *implementation* of the interface.

Defining a shared-memory multiprocessor's interface to memory is easier if we first consider a uniprocessor. A uniprocessor executes instructions and memory operations in a dynamic execution order called *program order*. Simple processors actually execute operations in program order while complex processors only appear to do so. In either case, each read must return the value of the last write to the same address, where *last* is uniquely defined by program order. If the uniprocessor is multiprogrammed, two cases exist. If a program executes as a single thread without sharing memory, then the programmer's memory interface is the same as for a uniprocessor without multiprogramming. The situation is more complex, on the other hand, if a program has multiple threads sharing memory (or the program shares memory with other running programs or is the operating system). In this case, the *last* write to an address could be by itself (the same thread) or by another thread (that was context switched onto the processor since this thread's last write to the address). In most cases, software uses synchronization to make program results meaningful.

Programmers can model a multiprogrammed uniprocessor as a merging of the program order of each executing thread into a single total order of processor execution. Most programmers, for example, would expect the code frag-

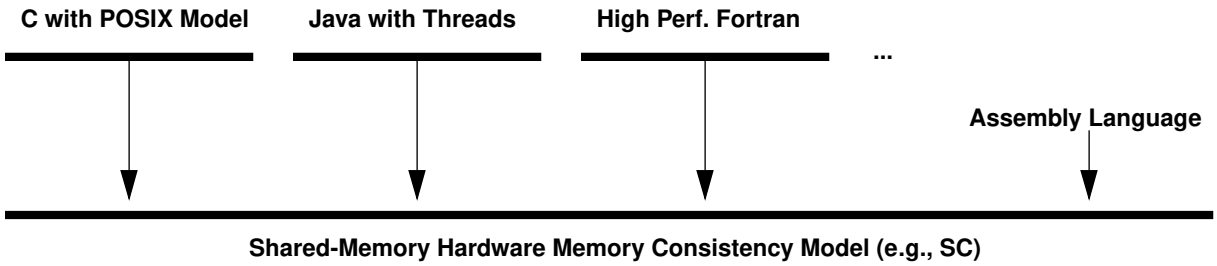

FIGURE 1. A shared memory multiprocessor has one or more high-level language (HLL) memory consistency models (upper lines) and one hardware memory consistency model (lower line). Middleware must preserve HLL memory semantics when translating programs to the hardware model. In contrast, assembly language programs are written directly to the hardware model.

ment in Table 1 (bottom of page 2) to set `data_copy` to the value of `new`.

Most computers today, however, are programmed in high-level languages (HLLs), such as C, C++, and Java. This practice creates two memory interface levels. At the higher level, each HLL defines memory for its programmers. At the lower level, hardware defines memory for low-level software, which I will call *middleware*. Middleware includes compilers, libraries, device drivers, and operating systems and some key parts of important applications (e.g., databases). For software written in HLLs, compilers must translate HLL memory operations into low-level ones in a manner that preserves memory semantics. In Table 1, for example, a compiler should not reorder P1's stores to `data` and `flag`. POSIX threads, for example, recommends that synchronization be encapsulated in library calls, such as `pthread_mutex_lock()`.

The interface for memory in a shared memory multiprocessor is called a *memory consistency model*. Similar to a uniprocessor, HLL programming induces the two levels of memory consistency models depicted in Figure 1: high-level models for each HLL and one low-level model for hardware. This paper is primarily concerned with hardware memory consistency models.

A multiprocessor can use the same relatively-simple memory interface as a multiprogrammed uniprocessor. This memory consistency model was formalized by Lamport as *sequential consistency* (SC) [8]. Section 2 argues the benefits of SC.

TABLE 1. Is `data_copy` always set to `new`?

Thread or Processor P1	Thread or Processor P2
`data = new;`	
`flag = SET;`	
	`while (flag != SET) {}`
	`data_copy = data;`

Perhaps surprisingly, the hardware memory consistency models of most commercial multiprocessors are not SC. This occurs because alternative models are believed to better facilitate high-performance implementations. Section 3 examines drawbacks of implementing SC and how alternative memory consistency models—called *relaxed models*—overcome some of them. Some of these models just relax the ordering from writes to reads (Section 3.1), while others aggressively relax the order among all normal reads and writes (Section 3.2). More details about these models and references to primary sources can be found in an excellent relaxed memory model tutorial by Adve and Gharachorloo [2], their Ph.D. theses [1,4], and Collier's tools for distinguishing memory models (`www.infomall.org/diagnostics/archtest.html`). For this reason, many academics, including myself, have advocated relaxed models over SC.

The advent of speculative execution has changed my mind. Section 4 argues that multiprocessor hardware should implement SC or, in some cases, models that just relax the ordering from writes to reads. I now see aggressively relaxed models as less attractive. I argue that the future performance gap between the aggressively relaxed models and SC will be in the range of 20% or less (Section 4.1). In my opinion, such a performance gap is not sufficient to justify burdening middleware authors with reasoning about aggressively relaxed memory models (Section 4.2). I also discuss alternatives, such as supporting SC with optional relaxed extensions or using commercial models that relax the order from writes to reads (especially when backward compatibility is involved).

2 Sequential Consistency

Lamport defined a multiprocessor to be *sequentially consistent* (SC) [8] 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 by its program.

The principle benefit to selecting SC as the interface to shared memory hardware is that it is what people expect. If you ask moderately sophisticated software professionals what shared memory does, they will likely define SC (albeit less precisely and less concisely than Lamport). Since good interfaces should not astonish their users, SC should be preferred.

A literal interpretation of SC's definition, however, might lead one to believe that implementing it requires one memory module and precludes per-processor caches. This is not the case. To the contrary, both SC and the relaxed models (described in Section 3) allow many optimizations important for high-performance implementations [6].

First, all models permit coherent caching. Caches may be private to processors or shared among some processors. They may be in one level or in multi-level hierarchies. A straightforward implementation of coherence processes operations of each processor in program order and does not perform a write until after it invalidates all other cached copies of the block.

Second, all models can use *non-binding prefetching*. Non-binding prefetching moves a block into a cache (in anticipation of use) but keeps the block under the jurisdiction of the coherence protocol. Non-binding prefetches affect memory performance but not memory semantics, because the prefetched block can be invalidated if another processor wishes to write to the block. (In contrast, a binding prefetch moves a datum into a register where it is unaffected by subsequent writes by other processors.) Non-binding prefetches can be initiated to overlap cache miss latency with computation or other misses by either hardware or software (through special prefetch instructions).

Third, all models can support *multithreading*, where a processor contains several "hot" threads (or processes) that it can run in an interleaved fashion. From a correctness point of view, a multiprocessor with *n k*-way multithreaded processors behaves like a multiprocessor with *n*×*k* conventional processors. The implementation of multithreaded hardware, however, can switch threads on long-latency events (e.g., to hide cache misses), can switch every cycle, or simultaneously execute multiple threads each cycle.

In summary, SC and the relaxed models allow all of the above optimizations. SC, however, permits the above optimizations and keeps the software interface simple. In particular, SC enables middleware authors to use the same target as a multiprogrammed uniprocessor. Thus, it seems hardware architects should choose SC.

3 Relaxed Models

Despite SC's advantages, most commercial hardware architectures have selected alternatives to SC called *relaxed* (or *weak*) *memory consistency models.* Relaxed models were selected to facilitate additional implementation optimizations whose use is precluded by SC without the complexity of speculative execution. SC, for example, makes it hard to use write buffers, because write buffers cause operations to be presented to the cache coherence protocol out of program order. Straightforward processors are also precluded from overlapping multiple reads and writes in the memory system. This restriction is crippling in systems without caches, where all operations go to memory. In systems with cache coherence, which are the norm today, this restriction impacts activity whenever operations miss or bypass the cache. (Cache bypassing occurs on uncacheable operations to input/output space, some block transfer operations, and writes to some coalescing write buffers.)

Definition for relaxed models are subtle and complex. I next discuss two important classes that Adve and Gharachorloo aptly call "*relaxing write to read program order*" and "*relaxing all orders*" [2].

3.1 Relaxing write to read program order

The class "*relaxing write to read program order*," sometimes called the *processor consistent* models, more-or-less exposes first-come-first-serve write buffers to low-level software. This means that a processor's writes may not immediately affect other processors, but when they do, the writes are seen in program order. If a processor does *write x*, *write flag*, and *read y*, for example, it can be sure that *x* is updated before *flag*, but it cannot know if either is done when it reads *y*. All common processors also make sure that a processor sees its own writes immediately (e.g., via write buffer bypassing). This ensures that these models have no effect on uniprocessor behavior.

Furthermore, the difference from SC makes no difference to most shared-memory programs, because most programs produce shared data by writing the data and then writing a flag or counter (e.g., Table 1). Programs only observe differences from SC in convoluted examples, like the code fragment illustrated in Table 2.

TABLE 2. PC, IBM 370, Pentium Pro and TSO allow both `x_copy` and `y_copy` to get old values, thereby violating SC.

Processor P1	Processor P2
`x = new;`	`y = new;`
`y_copy = y;`	`x_copy = x;`

Specific models in this class include *Wisconsin/Stanford processor consistency (PC), IBM 370, Intel Pentium Pro,*[1] and *Sun total store order (TSO)*. PC was proposed by Wisconsin and subsequently precisely defined by Stanford. In its full academic generality, PC is less useful than the others, because it does not guarantee a property called *causality*. Causality requires that *all other processors* see the effect of a processor's operation when *any other processor* sees it. Without causality, processor consistency can fail to look like SC in important cases involving three or more processors. One such case is illustrated in Table 3.

TABLE 3. "Causality" is needed to ensure `data_copy` is set to `new`

Processor P1	Processor P2	Processor P3
`data = new;`		
`flag1 = SET;`		
	`while (flag1 != SET){}`	
	`flag2 = SET;`	
		`while (flag2 != SET){}`
		`data_copy = data;`

The commercial models in this class (IBM 370, Pentium Pro, and TSO), on the other hand, guarantee causality and will always ensure that `data_copy` gets the value `new` in Table 3. Nevertheless, these models differ in other subtle ways, such as whether a processor reading its own write ensures that other processors also see it.

These hardware memory consistency models make it easier for hardware implementors to use many hardware optimizations found in uniprocessors. In particular, processor writes can be buffered in a first-come-first-serve write buffer in front of the cache and coherence protocol. Values of these buffered writes can often be bypassed to subsequent reads (by that processor to the same address) even before coherence permission has been obtained. This optimization is especially important for architectures with few general-purpose registers, such as the Intel Architecture-32.

Furthermore, in my opinion, having the hardware memory consistency model be IBM 370, Pentium Pro, TSO or processor consistency with causality has negligible impact on middleware authors. If these authors assume SC, they will rarely be astonished. These models look exactly like SC for the common idioms of data sharing (e.g, Table 1 and Table 3, but not Table 2).

1. I add the Pentium Pro memory model [7] (Section 7.2) to Adve and Gharachorloo's classification. Intel also states that the Pentium and Intel486 models are "virtually identical" to that of the Pentium Pro. Strictly speaking, a microprocessor must cooperate with a system to support a memory model. Thus, one can build systems with Pentium Pros that do not support the Pentium Pro memory model.

3.2 Relaxing all orders

The class "*relaxing all orders*" seeks to allow all the hardware implementation options of uniprocessors. Members of this class may completely reorder reads and writes and include *USC/Rice weak ordering (WO), Stanford release consistency (RC), DEC Alpha, IBM PowerPC,* and *Sun relaxed memory order (RMO)*. The models differ in subtle ways and in how programmers restore enough sanity to make examples like Table 1 behave as expected. WO and RC ask programmers to distinguish certain reads and writes as synchronization, so the hardware can handle these more carefully. The commercial models add special operations—variously called *fences*, *barriers, membars,* and *syncs*—to tell the system when order is required. Table 4 illustrates how the example in Table 1 could be augmented for Sun relaxed memory order. The `membar`

TABLE 4. Memory Barriers (`membars`) to ensure `data_copy` is always set to `new` under Sun RMO.

Processor P1	Processor P2
`data = new;`	
`membar #StoreStore`	
`flag = SET;`	
	`while (flag != SET) {}`
	`membar #LoadLoad`
	`data_copy = data;`

`#StoreStore` ensures `data` is written before `flag`, while `membar #LoadLoad` ensures `flag` is read before `data`. Commercial implementations of these models, however, make sure that a processor sees its own reads and writes in program order to preserve simple uniprocessor behavior.

Implementations of the models in this class can exploit many optimizations, because they need only implement order between operations when software asks for it and can be aggressively out-of-order the rest of the time. Processors can complete reads and writes to cache, for example, even while previous reads and writes (in program order) have not obtained coherence permission. With speculative execution, processors can *retire* reads and writes (i.e., preclude rollback) while previous reads and writes await coherence permission.

Furthermore, a hardware model from the "*relaxing all orders*" class does not appear to be too great a challenge for compiler writers. For sequential HLLs with threads, programmers often use synchronization libraries or declare critical variables `volatile`. In these cases, the compiler or library writer can add appropriate fences. For sequential languages with parallelizing compilers, the compiler

inserts the synchronization so it can know where the fences need to be.

In summary, relaxed models offer more hardware implementation options than SC and appear to use information that programmers or HLLs know anyway. Thus, it appears hardware should use relaxed models instead of SC.

4 Multiprocessors Should Implement SC

Section 4.1 discusses how future processors trends—especially more transistors and more speculation—affect the performance gap between relaxed and SC implementations. Section 4.2 explores the complexity of reasoning with relaxed models. Section 4.3 gives my opinion that the performance gain from relaxed models will not be sufficient to justify the intellectual complexity they add to the software/hardware interface.

4.1 The performance gap is not that large

The principal argument for relaxed models is that using them can yield higher performance than with SC. So what is the *performance gap* between relaxed models and SC? The answer is "it depends" on many processor implementation parameters.

Two observations by Gharachorloo et al. [5] have reduced the performance gap relative to initial expectations. First, SC hardware does not need to serialize the operations that obtain coherence permission (e.g., non-binding prefetches and cache misses). Instead, SC can overlap these operations just like relaxed implementations. SC implementations, however, should perform the actual reads and writes to and from the cache in program order. Thus, SC implementations can handle four cache misses on the sequence "*read A, write B, read C, write D*" in time modestly longer than handling one miss and three hits. Using a non-blocking cache, a SC implementation could pipeline "*get shared block A, get exclusive block B, get shared block C, get exclusive block D*" and then rapidly perform "*read A, write B, read C, write D*" as a series of cache hits.

Second, the advent of speculative execution allows both relaxed and SC implementations to be more aggressive. With speculative execution, a processor performs instructions eagerly. Sometimes instructions must be undone when speculation on previous instructions proves incorrect (e.g., mispredicted branches). A processor *commits* (or *retires*) an instruction when it is sure that an instruction will not need to be undone. Doing so usually frees up implementation resources. Instructions commit when (a) all previous instruction have committed and (b) the instruction's operation commits. A load or store operation commits when it is certain to read or write an appropriate memory value.

Speculative execution allows both relaxed and SC implementations to speculatively perform load and store operations out of order. In some cases, however, relaxed implementations can commit memory operations sooner than SC implementations. Consider, for example, a program that wishes to *read A* (which misses) and *read B* (which hits). Both relaxed and SC processors can perform the *read B* before even beginning the *read A*. Furthermore, relaxed processors can sometimes commit *read B* without waiting for *read A* to commit. SC processors, however, cannot commit *read B* until *read A* commits (or least obtains coherence permission for the block containing A). (Read B cannot be committed, because it may need to be unrolled if the block containing B must be invalidated due to an exclusive request by another processor before this processor obtains coherence permission to block A.) These sorts of techniques are already used in the HP PA-8000™, Intel Pentium Pro™ and MIPS R10000™.

While the speculative techniques are complex, their implication is simple:

> *Relaxed and SC implementations can do all the same speculations, but sometimes relaxed implementations can commit memory operations sooner.*

Thus, the performance gap between relaxed and SC implementations should narrow. The gap will be non-zero, however, if SC implementations (1) undo instructions more often or (2) more frequently exhaust implementation resources for uncommitted instructions. So quantitatively, what is the current performance gap?

The answer is, "it depends." It depends on benchmarks, memory latencies, and myriad of processor and cache implementation parameters. Ranganathan et al. [10, 9] provide an example of the academic state-of-the-art in 1997. They simulate a workload of six scientific benchmarks from Stanford and Rice on a MIPS R10000-like processor using 4-way instruction issue, dynamic scheduling with a 64-instruction window (instructions concurrently active), speculative execution, caches that support outstanding misses to up to eight distinct blocks, and many other assumptions that can be found in the paper.

I next consider four of their implementations. SC is an aggressive implementation of SC that uses hardware prefetching, speculative loads, and store buffering. Processor Consistency (PC) is an aggressive implementation that "*relaxes write to read program order*" and uses prefetching, speculative loads, and store buffering. Relaxed Consistency (RC) is an aggressive implementation that "*relaxes all orders*" and can use prefetching, speculative loads, and store buffering. I quote the RC version that has the best performance overall. This version disables prefetching and speculative loads whose overheads slightly surpass their

benefits (because RC already overlaps so many memory references). SC++ is an SC implementation that uses much more hardware and some new ideas in Ranganathan et al. [10].

Table 5 shows execution times for the six applications

TABLE 5. Execution Time Results from Ranganathan et al. [10].

Application	PC	RC	SC++
Erlebacher	99.1	92.8	92.5
Fft	85.2	81.7	85.9
Lu	94.4	94.3	90.2
Mp3d	91.0	70.1	95.5
Radix	76.4	62.8	87.0
Water	93.3	100.0	80.7
Average	**89.9**	**83.6**	**88.6**

normalized to the execution time for SC. PC improves on SC's execution by 0.8-23.6% while RC provides 0.0-37.2%. SC++ shows that the performance gap can be narrowed with much more hardware, but this comparison is unfair since PC and RC could also use more hardware.

What, however, is the performance gap for the whole workload? This number depends on how often and long each program is run. If one simplistically assumes that each program runs for a fixed amount of time under SC, then workload execution time under a model is the arithmetic average of program execution times. Doing this yields that PC and RC reduce execution time by 10% and 16%, respectively. These numbers correspond to performance improvements of 11% for PC and 20% for RC.

The performance gap on other workloads will be different and may be smaller. Relaxed models were designed for the instruction-level parallelism of scientific workloads, which tends to be larger than found for other workloads, such as operating systems and databases.

How will the performance gap change over the next ten years? One argument is that it will grow, because the latency to memory—measured in instruction issue opportunities—is likely to grow. On the contrary, I see two reasons that make it likely to shrink.

First, future microprocessor designers will be able to apply more transistors to enhance the effectiveness of known techniques for improving memory system performance. These techniques range from mundane measures like larger caches and more concurrent cache misses to sophisticated speculation and prefetching. Increasing the instruction window size, for example, will improve the performance of both SC and relaxed implementations by making instruction-window-full stalls less likely. The increased window size will also reduce the performance gap if the absolute difference in stalls gets smaller. This is likely due to the diminishing marginal utility of each additional instruction window buffer.

Second, architects will invent new microarchitectural techniques that, with speculation, can be applied to both SC and relaxed models. How can I be so confident? First, some of these techniques are already gestating, as can be found in a recent special issue of *IEEE Computer* [3] and in the annual proceedings of conferences like the *ACM/IEEE International Symposium on Computer Architecture* and *ACM International Symposium on Microarchitecture*. Second, architects in the past have always invented ways to innovatively "waste" a larger transistor budget. In 1996, Yu of Intel [12], re-examined Intel's 1989 predictions for 1996. He found that the predictions were accurate on technology (e.g., number of transistors per chip), but underestimated processor performance by a factor of 4 due to not anticipating the rapidity of microarchitecture innovation. I expect the innovation to continue (so I would not close the patent office).

If the performance gap is less than 20%, what will happen with relaxed models? Will middleware authors still find it worthwhile to program with relaxed models? The answer depends on how much burden it adds to middleware authors to make them reason with relaxed models.

4.2 Reasoning with relaxed models is hard

Before considering relaxed models, we need to consider the context. Authoring parallel middleware is hard. Many programming projects already stretch the intellectual limits of programmers to manage complexity while adding features, making behavior more robust, and staying on schedule. Dealing with relaxed models must necessarily either add a real cost (e.g., personnel or schedule delay) or opportunity cost (something else not done).

The costs of using relaxed models depends, in large part, on the complexity of reasoning with them. I find reasoning with relaxed models in the class "*relaxing all orders*" more difficult than reasoning with SC. Even though I have co-authored a half-dozen papers on the subject, I still have to think carefully before I can correctly make any precise statement about one of the existing models. Certainly middleware authors can understand the models, but do they want to spend their time dealing with definitions of various partial orders and about non-atomic operations? (A non-atomic operation allows its effects to be seen by some processors before others, in a manner detectable by running programs). Middleware authors must understand the models to a fairly good level of detail to be able to include sufficient fences without adding too many unnecessary ones. Too many unnecessary fences will reduce the performance gap seen in practice. In addition, authors of portable middleware (e.g., compilers) will need

to master different relaxed models for different hardware targets.

Others, however, will disagree with me and argue that reasoning with models in the class "*relaxing all orders*" is not that bad. Unfortunately, there is no final technical arbiter of whether something is "too complex." Thus, readers will have to decide for themselves.

What about hardware memory consistency models in the class "*relaxing write to read program order?*" In my opinion, middleware authors targeting these models have an easier task than those targeting the class "*relaxing all orders*." Assuming causality—as found in the commercial models of this class (IBM 370, Pentium Pro, and Sun TSO)—middleware authors can reason with SC and not have to consider placing fences, as long as they avoid using convoluted code (e.g., Table 2). On the other hand, compiler writers must still understand these models if they want to ensure that "convoluted code" behaves as written.

4.3 The bottom line

I recommend that future systems implement SC as their hardware memory consistency model. I do not believe that performance boost from implementing models in the class "*relaxing all orders*" is enough to justify the additional intellectual burden the relaxed models place on the middleware authors of complex commercial systems.

There are, however, several other viable alternatives. First, one can provide a first-class SC implementation and add optional relaxed support [4]. One could, for example, provide additional instructions that are more relaxed (e.g., Sun's block copy instructions) or multiple memory consistency model modes. Care must be exercised when adding options, however, because doing so incurs both implementation and verification costs. Multiple modes, in particular, can add significant verification costs if they enable a large new cross-product of hardware interactions.

Second, one can implement a model in the class "*relaxing write to read program order*" (that guarantees causality). These models allow hardware to play a few tricks more easily than with SC without affecting most middleware authors. (Woe, however, to those who are affected.). This option makes most sense for new implementations of existing systems that already "*relaxed write to read program order*." It can lead to subtle compatibility problems, however, if old systems provided SC. Third, one can implement a "*relaxing write to read program order*" model and add optional relaxed support.

5 Summary

Many future computers will contain multiple processors sharing memory. These hardware systems must define a memory consistency model to precisely define the allowable behavior of memory. A reasonable model is that a shared memory multiprocessor behaves like a multiprogrammed uniprocessor. This model was formalized by Lamport as sequential consistency (SC). Since SC appears to limit some implementation optimizations, researchers and practitioners have defined several relaxed models. Some of these models only relax the order of writes to reads (processor consistency, IBM 370, Pentium Pro, and TSO), while others aggressively relax order among normal reads and writes (WO, RC, Alpha, PowerPC, and RMO).

Nevertheless, I have argued that, with the advent of speculative execution, multiprocessor hardware should implement SC or, in some cases, models that just relax the ordering from writes to reads. I make a case that aggressively relaxed models are a less effective choice, because the future performance gap between the aggressively relaxed models and SC will not be sufficient to justify exposing the complexity of the aggressively relaxed models to the authors of low-level software.

While I argue that SC is preferred, several other viable alternatives also exist. First, one can support SC with optional relaxed extensions. Doing so can speed some operations, but pays implementation and verification costs regardless of whether the relaxed support is used in practice. Second, one can support a model like IBM 370, Pentium Pro, and TSO that allow hardware to play a few tricks more easily than with SC and appears like SC to almost all middleware. This option make most sense for new systems that must be backwardly compatible with old systems that use these models. Third, one can support one of these models with optional relaxed extensions.

Let me close by comparing instruction sets and hardware memory consistency models, two interfaces on the hardware/software boundary. Almost all current instruction sets present programmers and compilers with a sequential model (for each processor). Current implementations, however, now use complex pipelines, out-of-order execution and speculative execution to actually perform instructions out of program order, while at the same time using considerable logic to preserve the appearance of program order to software.

For the memory consistency model interface, we have a similar choice. With SC, we can hide the out-of-order complexity from software at some cost in implementation complexity. With relaxed models, complexity is made visible to the software interface. As with instruction sets, I think we should use SC to keep complexity off the interface and in the implementation where it belongs.

6 Acknowledgments

The ideas in this paper crystallized through interactions with many people at Wisconsin and during my 1995-1996 sabbatical at Sun Microsystems, which was graciously supported by *Greg Papadopoulos*. I thank the following peo-

ple—who may or may not agree with me—for their constructive comments on this paper: *Sarita Adve, Doug Burger, William Collier, Babak Falsafi, Kourosh Gharachorloo, Andy Glew, John Hennessy, Rebecca Hoffman, Alain Kägi, Shubu Mukherjee, Thomas Ruf, Guri Sohi, Jim Smith,* and *Dan Sorin*.

This work is supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525 and ARPA order no. B550, NSF Grants MIP-9225097 and MIPS-9625558, and donations from Sun Microsystems. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the U.S. Government.

References

[1] Sarita V. Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Computer Sciences Department, University of Wisconsin–Madison, November 1993.

[2] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[3] Douglas C. Burger and James R. Goodman (Editors). Special Issue on Billion-Transistor Architectures. *IEEE Computer*, 30(12), December 1997.

[4] Kourosh Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Computer System Laboratory, Stanford University, December 1995.

[5] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proceedings of the 1991 International Conference on Parallel Processing (Vol. I Architecture)*, pages I–355–364, August 1991.

[6] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 254–263, June 1991.

[7] Intel Corporation. *Pentium Pro Family Developer's Manual, Volume 3: Operating System Writer's Manual*, January 1996.

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

[9] Parthasarathy Ranganathan, Vijay S. Pai, Hazim Abdel-Shafi, and Sarita 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*, pages 144–156, June 1997.

[10] Parthasarathy Ranganathan, Vijay S. Pai, and Sarita V. Adve. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models. In *Proceedings of the Nineth ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 199–210, June 1997.

[11] David A. Wood and Mark D. Hill. Cost-Effective Parallel Computing. *IEEE Computer*, 28(2):69–72, February 1995.

[12] Albert Yu. The Future of Microprocessors. *IEEE Micro*, 16(6):46–53, December 1996.