Shared Memory Consistency Models: A Tutorial *

Sarita V. Adve† and Kourosh Gharachorloo‡

†Department of Electrical and Computer Engineering
Rice University
Houston, Texas 77251-1892

‡Western Research Laboratory
Digital Equipment Corporation
Palo Alto, California 94301-1616

Rice University ECE Technical Report 9512
Western Research Laboratory Research Report 95/7

September 1995

Abstract

Parallel systems that support the shared memory abstraction are becoming widely accepted in many areas of computing. Writing correct and efficient programs for such systems requires a formal specification of memory semantics, called a *memory consistency model*. The most intuitive model—*sequential consistency*—greatly restricts the use of many performance optimizations commonly used by uniprocessor hardware and compiler designers, thereby reducing the benefit of using a multiprocessor. To alleviate this problem, many current multiprocessors support more *relaxed consistency models*. Unfortunately, the models supported by various systems differ from each other in subtle yet important ways. Furthermore, precisely defining the semantics of each model often leads to complex specifications that are difficult to understand for typical users and builders of computer systems.

The purpose of this tutorial paper is to describe issues related to memory consistency models in a way that would be understandable to most computer professionals. We focus on consistency models proposed for hardware-based shared-memory systems. Many of these models are originally specified with an emphasis on the system optimizations they allow. We retain the system-centric emphasis, but use uniform and simple terminology to describe the different models. We also briefly discuss an alternate programmer-centric view that describes the models in terms of program behavior rather than specific system optimizations.[1]

*Most of this work was performed while Sarita Adve was at the University of Wisconsin-Madison and Kourosh Gharachorloo was at Stanford University. At Wisconsin, Sarita Adve was partly supported by an IBM graduate fellowship. She is currently supported by the National Science Foundation under Grant No. CCR-9502500 and CCR-9410457, by the Texas Advanced Technology Program under Grant No. 003604016, and by funds from Rice University. At Stanford, Kourosh Gharachorloo was supported by DARPA contract N00039-91-C-0138 and partly supported by a fellowship from Texas Instruments.

1 Introduction

The shared memory or single address space abstraction provides several advantages over the message passing (or private memory) abstraction by presenting a more natural transition from uniprocessors and by simplifying difficult programming tasks such as data partitioning and dynamic load distribution. For this reason, parallel systems that support shared memory are gaining wide acceptance in both technical and commercial computing.

To write correct and efficient shared memory programs, programmers need a precise notion of how memory behaves with respect to read and write operations from multiple processors. For example, consider the shared memory program fragment in Figure 1, which represents a fragment of the LocusRoute program from the SPLASH application suite. The figure shows processor P1 repeatedly allocating a task record, updating a data field within the record, and inserting the record into a task queue. When no more tasks are left, processor P1 updates a pointer, `Head`, to point to the first record in the task queue. Meanwhile, the other processors wait for `Head` to have a non-null value, dequeue the task pointed to by `Head` within a critical section, and finally access the data field within the dequeued record. What does the programmer expect from the memory system to ensure correct execution of this program fragment? One important requirement is that the value read from the data field within a dequeued record should be the same as that written by P1 in that record. However, in many commercial shared memory systems, it is possible for processors to observe the old value of the data field (i.e., the value prior to P1's write of the field), leading to behavior different from the programmer's expectations.

```
    Initially all pointers = null, all integers = 0.

P1                                      P2, P3, ..., Pn

while (there are more tasks) {          while (MyTask == null) {
  Task = GetFromFreeList();               Begin Critical Section
  Task → Data = ...;                      if (Head != null) {
  insert Task in task queue                 MyTask = Head;
}                                           Head = Head → Next;
Head = head of task queue;                }
                                          End Critical Section
                                        }
                                        ...  = MyTask → Data;
```

Figure 1: What value can a read return?

The *memory consistency model* of a shared-memory multiprocessor provides a formal specification of how the memory system will appear to the programmer, eliminating the gap between the behavior expected by the programmer and the actual behavior supported by a system. Effectively, the consistency model places restrictions on the values that can be returned by a read in a shared-memory program execution. Intuitively, a read should return the value of the "last" write to the same memory location. In uniprocessors, "last" is precisely defined by *program order*, i.e., the order in which memory operations appear in the program. This is not the case in multiprocessors. For example, in Figure 1, the write and read of the `Data` field within a record are not related by program order because they reside on two different processors. Nevertheless, an intuitive extension of the uniprocessor model can be applied to the multiprocessor case. This model is called *sequential consistency*. Informally, sequential consistency requires that all memory operations appear to execute one at a time, and the operations of a single processor appear to execute in the order described by that processor's program. Referring back to the program in Figure 1, this model ensures that the reads of the data field within a dequeued record will return the new values written by processor P1.

Sequential consistency provides a simple and intuitive programming model. However, it disallows many hardware and compiler optimizations that are possible in uniprocessors by enforcing a strict order among shared memory operations. For this reason, a number of more relaxed memory consistency models have been proposed, including some that are supported by commercially available architectures such as Digital Alpha, SPARC V8 and V9, and IBM PowerPC. Unfortunately, there has been a vast variety of relaxed consistency models proposed in the literature that differ from one another in subtle but important ways. Furthermore, the complex and non-uniform terminology that is used to describe these models makes it difficult to understand and compare them. This variety and complexity also often leads to misconceptions about relaxed memory consistency models, some of which are described in Figure 2.

The goal of this tutorial article is to provide a description of sequential consistency and other more relaxed memory consistency models in a way that would be understandable to most computer professionals. Such an understanding is important if the performance enhancing features that are being incorporated by system designers are to be correctly and widely used by programmers. To achieve this goal, we describe the semantics of different models using a simple and uniform terminology. We focus on consistency models proposed for hardware-based shared-memory systems. The original specifications of most of these models emphasized the system optimizations allowed by these models. We retain this system-centric emphasis in our descriptions to enable capturing the original semantics of the models. We also briefly describe an alternative, programmer-centric view of relaxed consistency models. This view describes models in terms of program behavior, rather than in terms of hardware or compiler optimizations. Readers interested in further pursuing a more formal treatment of both the system-centric and programmer-centric views may refer to our previous work [1, 6, 8].

The rest of this article is organized as follows. We begin with a short note on who should be concerned with the memory consistency model of a system. We next describe the programming model offered by sequential consistency, and the implications of sequential consistency on hardware and compiler implementations. We then describe several relaxed memory consistency models using a simple and uniform terminology. The last part of the article describes the programmer-centric view of relaxed memory consistency models.

2 Memory Consistency Models - Who Should Care?

As the interface between the programmer and the system, the effect of the memory consistency model is pervasive in a shared memory system. The model affects *programmability* because programmers must use it to reason about the correctness of their programs. The model affects the *performance* of the system because it determines the types of optimizations that may be exploited by the hardware and the system software. Finally, due to a lack of consensus on a single model, *portability* can be affected when moving software across systems supporting different models.

A memory consistency model specification is required for every level at which an interface is defined between the programmer and the system. At the machine code interface, the memory model specification affects the designer of the machine hardware and the programmer who writes or reasons about machine code. At the high level language interface, the specification affects the programmers who use the high level language and the designers of both the software that converts high-level language code into machine code and the hardware that executes this code. Therefore, the programmability, performance, and portability concerns may be present at several different levels.

In summary, the memory model influences the writing of parallel programs from the programmer's perspective, and virtually all aspects of designing a parallel system (including the processor, memory system, interconnection network, compiler, and programming languages) from a system designer's perspective.

3 Memory Semantics in Uniprocessor Systems

Most high-level uniprocessor languages present simple sequential semantics for memory operations. These semantics allow the programmer to assume that all memory operations will occur one at a time in the sequential order specified by the program (i.e., program order). Thus, the programmer expects a read will return the value of the last write to the same location before it by the sequential program order. Fortunately, the illusion of sequentiality can be supported efficiently. For example, it is sufficient to only maintain uniprocessor data and control dependences, i.e., execute two operations in program order when they are to the same location or when one controls the execution of the other. As long as these uniprocessor data and control dependences are respected, the compiler and hardware can freely reorder operations to different locations. This enables compiler optimizations such as register allocation, code motion, and loop transformations, and hardware optimizations, such as pipelining, multiple issue, write buffer bypassing and forwarding, and lockup-free caches, all of which lead to overlapping and reordering of memory operations. Overall, the sequential semantics of uniprocessors provide the programmer

Myth	Reality
A memory consistency model only applies to systems that allow multiple copies of shared data; e.g., through caching.	Figure 5 illustrates several counter-examples.
Most current systems are sequentially consistent.	Figure 9 mentions several commercial systems that are not sequentially consistent.
The memory consistency model only affects the design of the hardware.	The article describes how the memory consistency model affects many aspects of system design, including optimizations allowed in the compiler.
The relationship of cache coherence protocols to memory consistency models: (i) a cache coherence protocol inherently supports sequential consistency, (ii) the memory consistency model depends on whether the system supports an invalidate or update based coherence protocol.	The article discusses how the cache coherence protocol is only a part of the memory consistency model. Other aspects include the order in which a processor issues memory operations to the memory system, and whether a write executes atomically. The article also discusses how a given memory consistency model can allow both an invalidate or an update coherence protocol.
The memory model for a system may be defined solely by specifying the behavior of the processor (or the memory system).	The article describes how the memory consistency model is affected by the behavior of both the processor and the memory system.
Relaxed memory consistency models may not be used to hide read latency.	Many of the models described in this article allow hiding both read and write latencies.
Relaxed consistency models require the use of extra synchronization.	Most of the relaxed models discussed in this article do not require extra synchronization in the program. In particular, the programmer-centric framework only requires that operations be distinguished or labeled correctly. Other models provide safety nets that allow the programmer to enforce the required constraints for achieving correctness.
Relaxed memory consistency models do not allow chaotic (or asynchronous) algorithms.	The models discussed in this article allow chaotic (or asynchronous) algorithms. With system-centric models, the programmer can reason about the correctness of such algorithms by considering the optimizations that are enabled by the model. The programmer-centric approach simply requires the programmer to explicitly identify the operations that are involved in a race. For many chaotic algorithms, the former approach may provide higher performance since such algorithms do not depend on sequential consistency for correctness.

Figure 2: Some myths about memory consistency models.

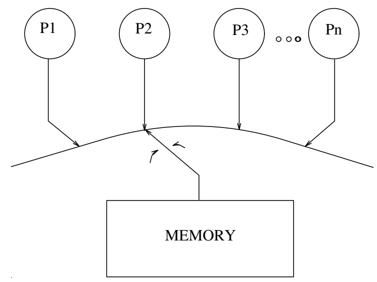


Figure 3: Programmer's view of sequential consistency.

with a simple and intuitive model and yet allow a wide range of efficient system designs.

4 Understanding Sequential Consistency

The most commonly assumed memory consistency model for shared memory multiprocessors is *sequential consistency*, formally defined by Lamport as follows [16].

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

There are two aspects to sequential consistency: (1) maintaining program order among operations from individual processors, and (2) maintaining a single sequential order among operations from all processors. The latter aspect makes it appear as if a memory operation executes *atomically* or *instantaneously* with respect to other memory operations.

Sequential consistency provides a simple view of the system to programmers as illustrated in Figure 3. Conceptually, there is a single global memory and a switch that connects an arbitrary processor to memory at any time step. Each processor issues memory operations in program order and the switch provides the global serialization among all memory operations.

Figure 4 provides two examples to illustrate the semantics of sequential consistency. Figure 4(a) illustrates the importance of program order among operations from a single processor. The code segment depicts an implementation of Dekker's algorithm for critical sections, involving two processors (P1 and P2) and two flag variables (`Flag1` and `Flag2`) that are initialized to 0. When P1 attempts to enter the critical section, it updates `Flag1` to 1, and checks the value of `Flag2`. The value 0 for `Flag2` indicates that P2 has not yet tried to enter the critical section; therefore, it is safe for P1 to enter. This algorithm relies on the assumption that a value of 0 returned by P1's read implies that P1's write has occurred before P2's write and read operations. Therefore, P2's read of the flag will return the value 1, prohibiting P2 from also entering the critical section. Sequential consistency ensures the above by requiring that program order among the memory operations of P1 and P2 be maintained, thus precluding the possibility of both processors reading the value 0 and entering the critical section.

Figure 4(b) illustrates the importance of atomic execution of memory operations. The figure shows three processors sharing variables `A` and `B`, both initialized to 0. Suppose processor P2 returns the value 1 (written by P1) for its read of `A`, writes to variable `B`, and processor P3 returns the value 1 (written by P2) for `B`. The atomicity aspect of sequential consistency allows us to assume the effect of P1's write is seen by the entire system at the same time. Therefore, P3 is guaranteed to see the effect of P1's write in the above execution and must return the value 1 for its read of `A` (since P3 sees the effect of P2's write after P2 sees the effect of P1's write to A).

5 Implementing Sequential Consistency

This section describes how the intuitive abstraction of sequential consistency shown in Figure 3 can be realized in a practical system. We will see that unlike uniprocessors, preserving the order of operations on a per-location basis

```
        Initially Flag1 = Flag2 = 0

P1                     P2

Flag1 = 1              Flag2 = 1
if (Flag2 == 0)        if (Flag1 == 0)
  critical section       critical section
```

(a)

```
      Initially A = B = 0

P1        P2          P3

A = 1
          if (A ==1)
            B = 1
                      if (B==1)
                        register1 = A
```

(b)

Figure 4: Examples for sequential consistency.

is not sufficient for maintaining sequential consistency in multiprocessors.

We begin by considering the interaction of sequential consistency with common hardware optimizations. To separate the issues of program order and atomicity, we first describe implementations of sequential consistency in architectures without caches and next consider the effects of caching shared data. The latter part of the section describes the interaction of sequential consistency with common compiler optimizations.

5.1 Architectures Without Caches

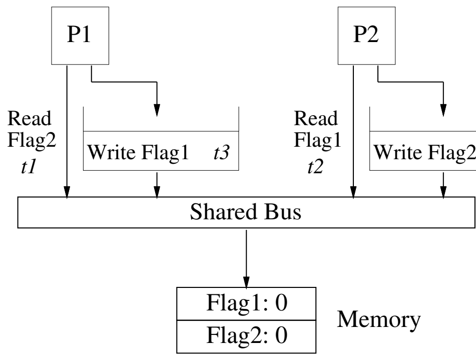
We have chosen three canonical hardware optimizations as illustrative examples of typical interactions that arise in implementing sequential consistency in the absence of data caching. A large number of other common hardware optimizations can lead to interactions similar to those illustrated by our canonical examples. As will become apparent, the key issue in correctly supporting sequential consistency in an environment without caches lies in maintaining the program order among operations from each processor. Figure 5 illustrates the various interactions discussed below. The terms *t1, t2, t3, ...* indicate the order in which the corresponding memory operations execute at memory.

5.1.1 Write Buffers with Bypassing Capability

The first optimization we consider illustrates the importance of maintaining program order between a write and a following read operation. Figure 5(a) shows an example bus-based shared-memory system with no caches. Assume a simple processor that issues memory operations one-at-a-time in program order. The only optimization we consider (compared to the abstraction of Figure 3) is the use of a write buffer with bypassing capability. On a write, a processor simply inserts the write operation into the write buffer and proceeds without waiting for the write to complete. Subsequent reads are allowed to bypass any previous writes in the write buffer for faster completion. This bypassing is allowed as long as the read address does not match the address of any of the buffered writes. The above constitutes a common hardware optimization used in uniprocessors to effectively hide the latency of write operations.

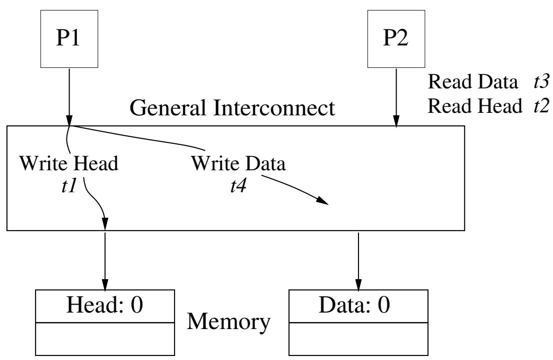
To see how the use of write buffers can violate sequential consistency, consider the program in Figure 5(a). The program depicts Dekker's algorithm also shown earlier in Figure 4(a). As explained earlier, a sequentially consistent system must prohibit an outcome where both the reads of the flags return the value 0. However, this outcome can occur in our example system. Each processor can buffer its write and allow the subsequent read to bypass the write in its write buffer. Therefore, both reads may be serviced by the memory system before either write is serviced, allowing both reads to return the value of 0.

The above optimization is safe in a conventional uniprocessor since bypassing (between operations to different locations) does not lead to a violation of uniprocessor data dependence. However, as our example illustrates, such a reordering can easily violate the semantics of sequential consistency in a multiprocessor environment.



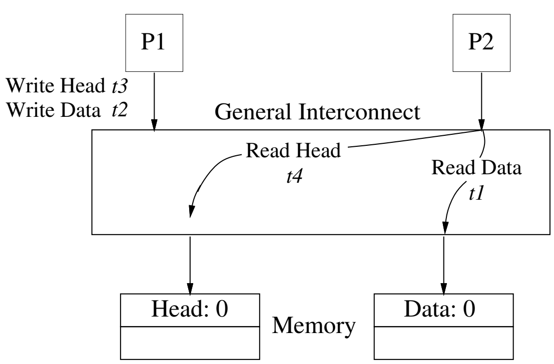
P1	P2
Flag1 = 1	Flag2 = 1
if (Flag2 == 0)	if (Flag1 == 0)
critical section	*critical section*

(a) write buffer



P1	P2
Data = 2000	while (Head == 0) {;}
Head = 1	... = Data

(b) overlapped writes



P1	P2
Data = 2000	while (Head == 0) {;}
Head = 1	... = Data

(c) non−blocking reads

Figure 5: Canonical optimizations that may violate sequential consistency.

5.1.2 Overlapping Write Operations

The second optimization illustrates the importance of maintaining program order between two write operations. Figure 5(b) shows an example system with a general (non-bus) interconnection network and multiple memory modules. A general interconnection network alleviates the serialization bottleneck of a bus-based design, and multiple memory modules provide the ability to service multiple operations simultaneously. We still assume processors issue memory operations in program order and proceed with subsequent operations without waiting for previous write operations to complete. The key difference compared to the previous example is that multiple write operations issued by the same processor may be simultaneously serviced by different memory modules.

The example program fragment in Figure 5(b) illustrates how the above optimization can violate sequential consistency; the example is a simplified version of the code shown in Figure 1. A sequentially consistent system guarantees that the read of `Data` by P2 will return the value written by P1. However, allowing the writes on P1 to be overlapped in the system shown in Figure 5(b) can easily violate this guarantee. Assume the `Data` and `Head` variables reside in different memory modules as shown in the figure. Since the write to `Head` may be injected into the network before the write to `Data` has reached its memory module, the two writes could complete out of program order. Therefore, it is possible for another processor to observe the new value of `Head` and yet obtain the old value of `Data`. Other common optimizations, such as coalescing writes to the same cache line in a write buffer (as in the Digital Alpha processors), can also lead to a similar reordering of write operations.

Again, while allowing writes to different locations to be reordered is safe for uniprocessor programs, the above example shows that such reordering can easily violate the semantics of sequential consistency. One way to remedy this problem is to wait for a write operation to reach its memory module before allowing the next write operation from the same processor to be injected into the network. Enforcing the above order typically requires an acknowledgement response for writes to notify the issuing processor that the write has reached its target. The acknowledgement response is also useful for maintaining program order from a write to a subsequent read in systems with general interconnection networks.

5.1.3 Non-Blocking Read Operations

The third optimization illustrates the importance of maintaining program order between a read and a following read or write operation. We consider supporting non-blocking reads in the system represented by Figure 5(b) and repeated in Figure 5(c). While most early RISC processors stall for the return value of a read operation (i.e., blocking read), many of the current and next generation processors have the capability to proceed past a read operation by using techniques such as non-blocking (lockup-free) caches, speculative execution, and dynamic scheduling.

Figure 5(c) shows an example of how overlapping reads from the same processor can violate sequential consistency. The program is the same as the one used for the previous optimization. Assume P1 ensures that its writes arrive at their respective memory modules in program order. Nevertheless, if P2 is allowed to issue its read operations in an overlapped fashion, there is the possibility for the read of `Data` to arrive at its memory module before the write from P1 while the read of `Head` reaches its memory module after the write from P1, which leads to a non-sequentially-consistent outcome. Overlapping a read with a following write operation can also present problems analogous to the above; this latter optimization is not commonly used in current processors, however.

5.2 Architectures With Caches

The previous section described complications that arise due to memory operation reordering when implementing the sequential consistency model in the absence of caches. Caching (or replication) of shared data can present similar reordering behavior that would violate sequential consistency. For example, a first level write through cache can lead to reordering similar to that allowed by a write buffer with bypassing capability, because reads that follow a write in program order may be serviced by the cache before the write completes. Therefore, an implementation with caches must also take precautions to maintain the illusion of program order execution for operations from each processor. Most notably, even if a read by a processor hits in the processor's cache, the processor typically cannot read the cached value until its previous operations by program order are complete.

The replication of shared data introduces three additional issues. First, the presence of multiple copies requires a mechanism, often referred to as the *cache coherence protocol*, to propagate a newly written value to all cached copies of the modified location. Second, detecting when a write is complete (to preserve program order between a write and its following operations) requires more transactions in the presence of replication. Third, propagating changes to multiple copies is inherently a non-atomic operation, making it more challenging to preserve the illusion of atomicity for writes with respect to other operations. We discuss each of these three issues in more detail below.

5.2.1 Cache Coherence and Sequential Consistency

Several definitions for cache coherence (also referred to as cache consistency) exist in the literature. The strongest definitions treat the term virtually as a synonym for sequential consistency. Other definitions impose extremely relaxed ordering guarantees. Specifically, one set of conditions commonly associated with a cache coherence protocol are: (1) a write is eventually made visible to all processors, and (2) writes to the same location appear to be seen in the same order by all processors (also referred to as *serialization* of writes to the same location) [13]. The above conditions are clearly not sufficient for satisfying sequential consistency since the latter requires writes to *all* locations (not just the same location) to be seen in the same order by all processors, and also explicitly requires that operations of a single processor appear to execute in program order.

We do not use the term cache coherence to define any consistency model. Instead, we view a cache coherence protocol simply as the mechanism that propagates a newly written value to the cached copies of the modified location. The propagation of the value is typically achieved by either *invalidating* (or eliminating) the copy or *updating* the copy to the newly written value. With this view of a cache coherence protocol, a memory consistency model can be interpreted as the policy that places an early and late bound on when a new value can be propagated to any given processor.

5.2.2 Detecting the Completion of Write Operations

As mentioned in the previous section, maintaining the program order from a write to a following operation typically requires an acknowledgement response to signal the completion of the write. In a system without caches, the acknowledgement response may be generated as soon as the write reaches its target memory module. However, the above may not be sufficient in designs with caches. Consider the code in Figure 5(b), and a system similar to the one depicted in the same figure but enhanced with a write through cache for each processor. Assume that processor P2 initially has `Data` in its cache. Suppose P1 proceeds with its write to `Head` after its previous write to `Data` reaches its target memory but before its value has been propagated to P2 (via an invalidation or update message). It is now possible for P2 to read the new value of `Head` and still return the old value of `Data` from its cache, a violation of sequential consistency. This problem can be avoided if P1 waits for P2's cache copy of `Data` to be updated or invalidated before proceeding with the write to `Head`.

Therefore, on a write to a line that is replicated in other processor caches, the system typically requires a mechanism to acknowledge the receipt of invalidation or update messages by the target caches. Furthermore, the acknowledgement messages need to be collected (either at the memory or at the processor that issues the write), and the processor that issues the write must be notified of their completion. A processor can consider a write to be complete only after the above notification. A common optimization is to acknowledge the invalidation or update message as soon as it is received by a processing node and potentially before the actual cache copy is affected; such a design can still satisfy sequential consistency as long as certain ordering constraints are observed in processing the incoming messages to the cache [6].

5.2.3 Maintaining the Illusion of Atomicity for Writes

While sequential consistency requires memory operations to appear atomic or instantaneous, propagating changes to multiple cache copies is inherently a non-atomic operation. We motivate and describe two conditions that can together ensure the appearance of atomicity in the presence of data replication. The problems due to non-atomicity are easier to illustrate with with update-based protocols; therefore, the following examples assume such a protocol.

To motivate the first condition, consider the program in Figure 6. Assume all processors execute their memory operations in program order and one-at-a-time. It is possible to violate sequential consistency if the updates for the writes of A by processors P1 and P2 reach processors P3 and P4 in a different order. Thus, processors P3

Initially A = B = C = 0

P1	P2	P3	P4
A = 1	A = 2	while (B != 1) {;}	while (B != 1) {;}
B = 1	C = 1	while (C != 1) {;}	while (C != 1) {;}
		register1 = A	register2 = A

Figure 6: Example for serialization of writes.

and P4 can return different values for their reads of A (e.g., register1 and register2 may be assigned the values 1 and 2 respectively), making the writes of A appear non-atomic. The above violation of sequential consistency is possible in systems that use a general interconnection network (e.g., Figure 5(b)), where messages travel along different paths in the network and no guarantees are provided on the order of delivery. The violation can be avoided by imposing the condition that writes to the *same* location be serialized; i.e., all processors see writes to the same location in the same order. Such serialization can be achieved if all updates or invalidates for a given location originate from a single point (e.g., the directory) and the ordering of these messages between a given source and destination is preserved by the network. An alternative is to delay an update or invalidate from being sent out until any updates or invalidates that have been issued on behalf of a previous write to the same location are acknowledged.

To motivate the second condition, consider the program fragment in Figure 4(b), again with an update protocol. Assume all variables are initially cached by all processors. Furthermore, assume all processors execute their memory operations in program order and one-at-a-time (waiting for acknowledgements as described above), and writes to the same location are serialized. It is still possible to violate sequential consistency on a system with a general network if (1) processor P2 reads the new value of `A` before the update of `A` reaches processor P3, (2) P2's update of `B` reaches P3 before the update of `A`, and (3) P3 reads the new value of `B` and then proceeds to read the value of `A` from its own cache (before it gets P1's update of `A`). Thus, P2 and P3 appear to see the write of A at different times, making the write appear non-atomic. An analogous situation can arise in an invalidation-based scheme.

The above violation of sequential consistency occurs because P2 is allowed to return the value of the write to A before P3 has seen the update generated by this write. One possible restriction that prevents such a violation is to prohibit a read from returning a newly written value until all cached copies have acknowledged the receipt of the invalidation or update messages generated by the write. This condition is straightforward to ensure with invalidation-based protocols. Update-based protocols are more challenging because unlike invalidations, updates directly supply new values to other processors. One solution is to employ a two phase update scheme. The first phase involves sending updates to the processor caches and receiving acknowledgements for these updates. In this phase, no processor is allowed to read the value of the updated location. In the second phase, a confirmation message is sent to the updated processor caches to confirm the receipt of all acknowledgements. A processor can use the updated value from its cache once it receives the confirmation message from the second phase. However, the processor that issued the write can consider its write complete at the end of the first phase.

5.3 Compilers

The interaction of the program order aspect of sequential consistency with the compiler is analogous to that with the hardware. Specifically, for all the program fragments discussed so far, compiler-generated reordering of shared memory operations will lead to violations of sequential consistency similar to hardware-generated reorderings. Therefore, in the absence of more sophisticated analysis, a key requirement for the compiler is to preserve program order among shared memory operations. This requirement directly restricts any uniprocessor compiler optimization that can result in reordering memory operations. These include simple optimizations such as code motion, register allocation, and common sub-expression elimination, and more sophisticated optimizations such as loop blocking or software pipelining.

In addition to a reordering effect, optimizations such as register allocation also lead to the elimination of certain shared memory operations that can in turn violate sequential consistency. Consider the code in Figure 5(b). If the compiler register allocates the location `Head` on P2 (by doing a single read of `Head` into a register and then reading the value within the register), the loop on P2 may never terminate in some executions (if the single read on P2 returns the old value of `Head`). However, the loop is guaranteed to terminate in every sequentially consistent execution of the code. The source of the problem is that allocating `Head` in a register on P2 prohibits P2 from ever observing the new value written by P1.

In summary, the compiler for a shared memory parallel program can not directly apply many common optimizations used in a uniprocessor compiler if sequential consistency is to be maintained. The above comments apply to compilers for explicitly parallel programs; compilers that parallelize sequential code naturally have enough information about the resulting parallel program they generate to determine when optimizations can be safely applied.

5.4 Summary for Sequential Consistency

From the above discussion, it is clear that sequential consistency constrains many common hardware and compiler optimizations. Straightforward hardware implementations of sequential consistency typically need to satisfy the following two requirements. *First*, a processor must ensure that its previous memory operation is complete before proceeding with its next memory operation in program order. We call this requirement the *program order* requirement. Determining the completion of a write typically requires an explicit acknowledgement message from memory. Additionally, in a cache-based system, a write must generate invalidate or update messages for all cached copies, and the write can be considered complete only when the generated invalidates and updates are acknowledged by the target caches. The *second* requirement pertains only to cache-based systems and concerns write atomicity. It requires that writes to the same location be serialized (i.e., writes to the same location be made visible in the same order to all processors) and that the value of a write not be returned by a read until all invalidates or updates generated by the write are acknowledged (i.e., until the write becomes visible to all processors). We call this the *write atomicity* requirement. For compilers, an analog of the program order requirement applies to straightforward implementations. Furthermore, eliminating memory operations through optimizations such as register allocation can also violate sequential consistency.

A number of techniques have been proposed to enable the use of certain optimizations by the hardware and compiler without violating sequential consistency; those having the potential to substantially boost performance are discussed below.

We first discuss two hardware techniques applicable to sequentially consistent systems with hardware support for cache coherence [10]. The first technique automatically prefetches ownership for any write operations that are delayed due to the program order requirement (e.g., by issuing prefetch-exclusive requests for any writes delayed in the write buffer), thus partially overlapping the service of the delayed writes with the operations preceding them in program order. This technique is only applicable to cache-based systems that use an invalidation-based protocol. The second technique speculatively services read operations that are delayed due to the program order requirement; sequential consistency is guaranteed by simply rolling back and reissuing the read and subsequent operations in the infrequent case that the read line gets invalidated or updated before the read could have been issued in a more straightforward implementation. This latter technique is suitable for dynamically scheduled processors since much of the roll back machinery is already present to deal with branch mispredictions. The above two techniques will be supported by several next generation microprocessors (e.g., MIPS R10000, Intel P6), thus enabling more efficient hardware implementations of sequential consistency.

Other latency hiding techniques, such as non-binding software prefetching or hardware support for multiple contexts, have been shown to enhance the performance of sequentially consistent hardware. However, the above techniques are also beneficial when used in conjunction with relaxed memory consistency.

Finally, Shasha and Snir developed a compiler algorithm to detect when memory operations can be reordered without violating sequential consistency [18]. Such an analysis can be used to implement both hardware and compiler optimizations by reordering only those operation pairs that have been analyzed to be safe for reordering by the compiler. The algorithm by Shasha and Snir has exponential complexity [15]; more recently, a new algorithm has been proposed for SPMD programs with polynomial complexity [15]. However, both algorithms require global dependence analysis to determine if two operations from different processors can conflict (similar to alias analysis);

this analysis is difficult and often leads to conservative information which can decrease the effectiveness of the algorithm.

It remains to be seen if the above hardware and compiler techniques can approach the performance of more relaxed consistency models. The remainder of this article focuses on relaxing the memory consistency model to enable many of the optimizations that are constrained by sequential consistency.

6 Relaxed Memory Models

As an alternative to sequential consistency, several relaxed memory consistency models have been proposed in both academic and commercial settings. The original descriptions for most of these models are based on widely varying specification methodologies and levels of formalism. The goal of this section is to describe these models using simple and uniform terminology. The original specifications of these models emphasized system optimizations enabled by the models; we retain the system-centric emphasis in our descriptions of this section. We focus on models proposed for hardware shared-memory systems; relaxed models proposed for software-supported shared-memory systems are more complex to describe and beyond the scope of this paper. A more formal and unified system-centric framework to describe both hardware and software based models, along with a formal description of several models within the framework, appears in our previous work [8, 6].

We begin this section by describing the simple methodology we use to characterize the various models, and then describe each model using this methodology.

6.1 Characterizing Different Memory Consistency Models

We categorize relaxed memory consistency models based on two key characteristics: (1) how they relax the program order requirement, and (2) how they relax the write atomicity requirement.

With respect to program order relaxations, we distinguish models based on whether they relax the order from a write to a following read, between two writes, and finally from a read to a following read or write. In all cases, the relaxation only applies to operation pairs with different addresses. These relaxations parallel the optimizations discussed in Section 5.1.

With respect to the write atomicity requirement, we distinguish models based on whether they allow a read to return the value of *another* processor's write before all cached copies of the accessed location receive the invalidation or update messages generated by the write; i.e., before the write is made visible to all other processors. This relaxation was described in Section 5.2 and only applies to cache-based systems.

Finally, we consider a relaxation related to both program order and write atomicity, where a processor is allowed to read the value of its own previous write before the write is made visible to other processors. In a cache-based system, this relaxation allows the read to return the value of the write before the write is serialized with respect to other writes to the same location and before the invalidations/updates of the write reach any other processor. An example of a common optimization that is allowed by this relaxation is forwarding the value of a write in a write buffer to a following read from the same processor. For cache-based systems, another common example is where a processor writes to a write-through cache, and then reads the value from the cache before the write is complete. We consider this relaxation separately because it can be safely applied to many of the models without violating the semantics of the model, even though several of the models do not explicitly specify this optimization in their original definitions. For instance, this relaxation is allowed by sequential consistency as long as all other program order and atomicity requirements are maintained [8], which is why we did not discuss it in the previous section. Furthermore, this relaxation can be safely applied to all except one of the models discussed in this section.

Figure 7 summarizes the relaxations discussed above. Relaxed models also typically provide programmers with mechanisms for overriding such relaxations. For example, explicit *fence* instructions may be provided to override program order relaxations. We generically refer to such mechanisms as the *safety net* for a model, and will discuss the types of safety nets provided by each model. Each model may provide more subtle ways of enforcing specific ordering constraints; for simplicity, we will only discuss the more straightforward safety nets.

Figure 8 provides an overview of the models described in the remaining part of this section. The figure shows whether a straightforward implementation of the model can efficiently exploit the program order or write atomicity

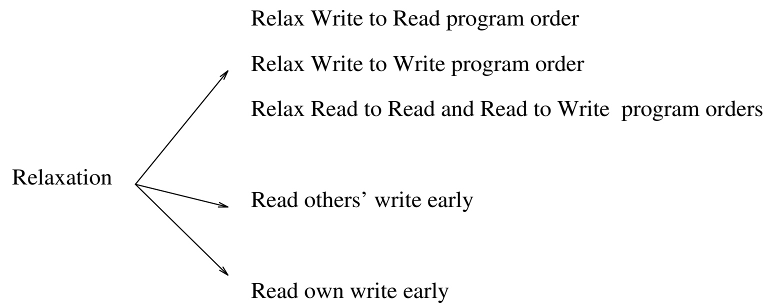


Figure 7: Relaxations allowed by memory models. The first three (program order) relaxations apply only to operation pairs accessing different locations.

Relaxation	W → R Order	W → W Order	R → RW Order	Read Others' Write Early	Read Own Write Early	Safety net
SC [16]					√	
IBM 370 [14]	√					serialization instructions
TSO [20]	√				√	RMW
PC [13, 12]	√			√	√	RMW
PSO [20]	√	√			√	RMW, STBAR
WO [5]	√	√	√		√	synchronization
RCsc [13, 12]	√	√	√		√	release, acquire, nsync, RMW
RCpc [13, 12]	√	√	√	√	√	release, acquire, nsync, RMW
Alpha [19]	√	√	√		√	MB, WMB
RMO [21]	√	√	√		√	various MEMBAR's
PowerPC [17, 4]	√	√	√	√	√	SYNC

Figure 8: Simple categorization of relaxed models. A √ indicates that the corresponding relaxation is allowed by straightforward implementations of the corresponding model. It also indicates that the relaxation can be detected by the programmer (by affecting the results of the program) except for the following cases. The "Read Own Write Early" relaxation is not detectable with the SC, WO, Alpha, and PowerPC models. The "Read Others' Write Early" relaxation is possible and detectable with complex implementations of RCsc.

Relaxation	Example Commercial Systems Providing the Relaxation
W → R Order	AlphaServer 8200/8400, Cray T3D, Sequent Balance, SparcCenter1000/2000
W → W Order	AlphaServer 8200/8400, Cray T3D
R → RW Order	AlphaServer 8200/8400, Cray T3D
Read Others' Write Early	Cray T3D
Read Own Write Early	AlphaServer 8200/8400, Cray T3D, SparcCenter1000/2000

Figure 9: Some commercial systems that relax sequential consistency.

relaxations described above, and mentions the safety nets provided by each model. The figure also indicates when the above relaxations are detectable by the programmer; i.e., when they can affect the results of the program. Figure 9 gives examples of commercial systems that allow the above relaxations. For simplicity, we do not attempt to describe the semantics of the models with respect to issues such as instruction fetches or multiple granularity operations (e.g., byte and word operations) even though such semantics are defined by some of these models.

The following sections describe each model in more detail and discuss the implications of each model on hardware and compiler implementations. Throughout this discussion, we implicitly assume that the following constraints are satisfied. First, we assume all models require a write to eventually be made visible to all processors and for writes to the same location to be serialized. These requirements are trivially met if shared data is not cached, and are usually met by a hardware cache coherence protocol in the presence of shared data caching. Second, we assume all models enforce uniprocessor data and control dependences. Finally, models that relax the program order from reads to following write operations must also maintain a subtle form of multiprocessor data and control dependences [8, 1]; this latter constraint is inherently upheld by all processor designs we are aware of and can also be easily maintained by the compiler.

6.2 Relaxing the *Write* to *Read* Program Order

The first set of models we discuss relax the program order constraints in the case of a write followed by a read to a different location. These models include the IBM 370 model, the SPARC V8 total store ordering model (TSO), and the processor consistency model (PC) (this differs from the processor consistency model defined by Goodman).

The key program order optimization enabled by these models is to allow a read to be reordered with respect to previous writes from the same processor. As a consequence of this reordering, programs such as the one in Figure 5(a) can fail to provide sequentially consistent results. However, the violations of sequential consistency illustrated in Figure 5(b) and Figure 5(c) cannot occur due to the enforcement of the remaining program order constraints.

The three models differ in when they allow a read to return the value of a write. The IBM 370 model is the strictest because it prohibits a read from returning the value of a write before the write is made visible to all processors. Therefore, even if a processor issues a read to the same address as a previous pending write from itself, the read must be delayed until the write is made visible to all processors. The TSO model partially relaxes the above requirement by allowing a read to return the value of its own processor's write even before the write is serialized with respect to other writes to the same location. However, as with sequential consistency, a read is not allowed to return the value of another processor's write until it is made visible to all other processors. Finally, the PC model relaxes both constraints, such that a read can return the value of any write before the write is serialized or made visible to other processors. Figure 10 shows example programs that illustrate these differences among the above three models.

We next consider the safety net features for the above three models. To enforce the program order constraint from a write to a following read, the IBM 370 model provides special *serialization instructions* that may be placed between the two operations. Some serialization instructions are special memory instructions that are used for synchronization (e.g., compare&swap), while others are non-memory instructions such as a branch. Referring back to the example program in Figure 5(a), placing a serialization instruction after the write on each processor provides sequentially consistent results for the program even when it is executed on the IBM 370 model.

In contrast to IBM 370, the TSO and PC models do not provide explicit safety nets. Nevertheless, programmers can use read-modify-write operations to provide the illusion that program order is maintained between a write and a following read. For TSO, program order appears to be maintained if either the write or the read is already part of a read-modify-write or is replaced by a read-modify-write. To replace a read with a read-modify-write, the write in the read-modify-write must be a "dummy" write that writes back the read value. Similarly, replacing a write with a read-modify-write requires writing back the desired value regardless of what the read returns. Therefore, the above techniques are only applicable in designs that provide such flexibility for read-modify-write instructions. For PC, program order between a write and a following read appears to be maintained if the read is replaced by or is already part of a read-modify-write. In contrast to TSO, replacing the write with a read-modify-write is not sufficient for imposing this order in PC. The difference arises because TSO places more stringent constraints on the behavior of read-modify-writes; specifically, TSO requires that no other writes to any location appear to occur between the read and the write of the read-modify-write, while PC requires this for writes to the same location only.

```
Initially A = Flag1 = Flag2 = 0

P1                     P2

Flag1 = 1              Flag2 = 1
A = 1                  A = 2
register1 = A          register3 = A
register2 = Flag2      register4 = Flag1

Result: register1 = 1, register3 = 2,
        register2 = register4 = 0
```

(a)

```
Initially A = B = 0

P1        P2              P3

A = 1
          if (A == 1)
             B = 1
                          if (B == 1)
                             register1 = A

Result: B = 1, register1 = 0
```

(b)

Figure 10: Differences between 370, TSO, and PC. The result for the program in part (a) is possible with TSO and PC because both models allow the reads of the flags to occur before the writes of the flags on each processor. The result is not possible with IBM 370 because the read of A on each processor is not issued until the write of A on that processor is done. Consequently, the read of the flag on each processor is not issued until the write of the flag on that processor is done. The program in part (b) is the same as in Figure 4(b). The result shown is possible with PC because it allows P2 to return the value of P1's write before the write is visible to P3. The result is not possible with IBM 370 or TSO.

We next consider the safety net for enforcing the atomicity requirement for writes. IBM 370 does not need a safety net since it does not relax atomicity. For TSO, a safety net for write atomicity is required only for a write that is followed by a read to the same location in the same processor; the atomicity can be achieved by ensuring program order from the write to the read using read-modify-writes as described above. For PC, a write is guaranteed to appear atomic if every read that may return the value of the write is part of, or replaced with, a read-modify-write.

The reasoning for how read-modify-write operations ensure the required program order or atomicity in the above models is beyond the scope of this paper [7]. There are some disadvantages to relying on a read-modify-write as a safety net in models such as TSO and PC. First, a system may not implement a general read-modify-write that can be used to appropriately replace any read or write. Second, replacing a read by a read-modify-write incurs the extra cost of performing the write (e.g., invalidating other copies of the line). Of course, these safety nets do not add any overhead if the specific read or write operations are already part of read-modify-write operations. Furthermore, most programs do not frequently depend on the write to read program order or write atomicity for correctness.

Relaxing the program order from a write followed by a read can improve performance substantially at the hardware level by effectively hiding the latency of write operations [9]. For compiler optimizations, however, this relaxation alone is not beneficial in practice. The reason is that reads and writes are usually finely interleaved in a program; therefore, most reordering optimizations effectively result in reordering with respect to both reads and writes. Thus, most compiler optimizations require the full flexibility of reordering any two operations in program order; the ability to only reorder a write with respect to a following read is not sufficiently flexible.

6.3 Relaxing the *Write* to *Read* and *Write* to *Write* Program Orders

The second set of models further relax the program order requirement by eliminating ordering constraints between writes to different locations. The SPARC V8 partial store ordering model (PSO) is the only example of such a model that we describe here. The key additional hardware optimization enabled by PSO over the previous set of models is that writes to different locations from the same processor can be pipelined or overlapped and are allowed to reach memory or other cached copies out of program order. With respect to atomicity requirements, PSO is identical to TSO by allowing a processor to read the value of its own write early, and prohibiting a processor from reading the value of another processor's write before the write is visible to all other processors. Referring back to the programs in Figures 5(a) and (b), PSO allows non-sequentially consistent results.

The safety net provided by PSO for imposing the program order from a write to a read, and for enforcing write atomicity, is the same as TSO. PSO provides an explicit *STBAR* instruction for imposing program order between two writes. One way to support a STBAR in an implementation with FIFO write buffers is to insert the STBAR in the write buffer, and delay the retiring of writes that are buffered after a STBAR until writes that were buffered before the STBAR have retired and completed. A counter can be used to determine when all writes before the STBAR have completed—a write sent to the memory system increments the counter, a write acknowledgement decrements the counter, and the counter value 0 indicates that all previous writes are complete. Referring back to the program in Figure 5(b), inserting a STBAR between the two writes ensures sequentially consistent results with PSO.

As with the previous set of models, the optimizations allowed by PSO are not sufficiently flexible to be useful to a compiler.

6.4 Relaxing All Program Orders

The final set of models we consider relax program order between all operations to different locations. Thus, a read or write operation may be reordered with respect to a following read or write to a different location. We discuss the weak ordering (WO) model, two flavors of the release consistency model (RCsc/RCpc), and three models proposed for commercial architectures: the Digital Alpha, SPARC V9 relaxed memory order (RMO), and IBM PowerPC models. Except for Alpha, the above models also allow the reordering of two reads to the same location. Referring back to Figure 5, the above models violate sequential consistency for all the code examples shown in the figure.

The key additional program order optimization allowed relative to the previous models is that memory operations following a read operation may be overlapped or reordered with respect to the read operation. In hardware, this flexibility provides the possibility of hiding the latency of read operations by implementing true non-blocking reads in the context of either static (in-order) or dynamic (out-of-order) scheduling processors, supported by techniques such as non-blocking (lockup-free) caches and speculative execution [11].

All of the models in this group allow a processor to read its own write early. However, RCpc and PowerPC are the only models whose straightforward implementations allow a read to return the value of another processor's write early. It is possible for more complex implementations of WO, RCsc, Alpha, and RMO to achieve the above. From the programmer's perspective, however, all implementations of WO, Alpha, and RMO must preserve the illusion of write atomicity.[2] RCsc is a unique model in this respect; programmers cannot rely on atomicity since complex implementations of RCsc can potentially violate atomicity in a way that can affect the result of a program.

The above models may be separated into two categories based on the type of safety net provided. The WO, RCsc, and RCpc models distinguish memory operations based on their type, and provide stricter ordering constraints for some types of operations. On the other hand, the Alpha, RMO, and PowerPC models provide explicit fence instructions for imposing program orders between various memory operations. The following describes each of these models in greater detail, focusing on their safety nets. Implications for compiler implementations for the models in this group are discussed at the end of this section.

6.4.1 Weak Ordering (WO)

The weak ordering model classifies memory operations into two categories: *data* operations and *synchronization* operations. To enforce program order between two operations, the programmer is required to identify at least one of the operations as a synchronization operation. This model is based on the intuition that reordering memory operations to data regions between synchronization operations does not typically affect the correctness of a program.

Operations distinguished as synchronization effectively provide a safety net for enforcing program order. We briefly describe a simple way to support the appropriate functionality in hardware. Each processor can provide a counter to keep track of its outstanding operations. This counter is incremented when the processor issues an operation and is decremented when a previously issued operation completes. Each processor must ensure that a synchronization operation is not issued until all previous operations are complete, which is signaled by a zero value

[2] For WO, given a read R followed by a write W in program order that are related by the multiprocessor data or control dependence (mentioned in Section 6.1), we assume the write W is delayed until both the read R is complete and the write that is read by R is complete.

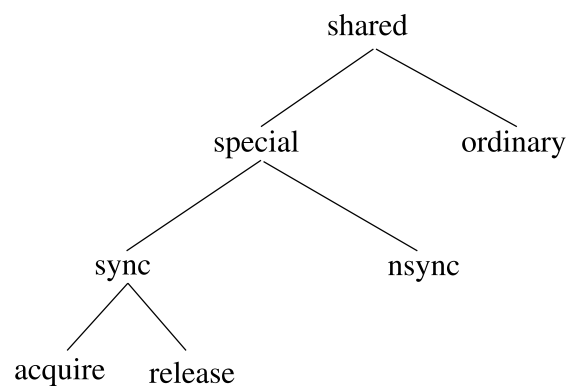


Figure 11: Distinguishing operations for release consistency.

for the counter. Furthermore, no operations are issued until the previous synchronization operation completes. Note that memory operations between two synchronization operations may still be reordered and overlapped with respect to one another.

The weak ordering model ensures that writes always appear atomic to the programmer; therefore, no safety net is required for write atomicity.

6.4.2 Release Consistency (RCsc/RCpc)

Compared to weak ordering, release consistency provides further distinctions among memory operations. Figure 11 pictorially depicts this classification of memory operations. Operations are first distinguished as *ordinary* or *special*. These two categories loosely correspond to the data and synchronization categories in WO. Special operations are further distinguished as *sync* or *nsync* operations. Syncs intuitively correspond to synchronization operations, whereas nsyncs correspond to asynchronous data operations or special operations that are not used for synchronization. Finally, sync operations are further distinguished as *acquire* or *release* operations. Intuitively, an acquire is a read memory operation that is performed to gain access to a set of shared locations (e.g., a lock operation or spinning for a flag to be set). A release is a write operation that is performed to grant permission for accessing a set of shared locations (e.g., an unlock operation or setting of a flag).

There are two flavors of release consistency that differ based on the program orders they maintain among special operations. The first flavor maintains sequential consistency among special operations (RCsc), while the second flavor maintains processor consistency among such operations (RCpc). Below, we depict the program order constraints for these two models for operations to different locations. In our notation, $A \rightarrow B$ implies that if operation type A precedes operation type B in program order, then program order is enforced between the two operations. For RCsc, the constraints are as follows:

- acquire $\rightarrow$ all, all $\rightarrow$ release, and special $\rightarrow$ special.

For RCpc, the write to read program order among special operations is eliminated:

- acquire $\rightarrow$ all, all $\rightarrow$ release, and special $\rightarrow$ special except for a special write followed by a special read.

Therefore, enforcing program order between a pair of operations can be achieved by distinguishing or labeling appropriate operations based on the above information. For RCpc, imposing program order from a write to a read operation requires using read-modify-write operations analogous to the PC model. Further, if the write being ordered is ordinary, then the write in the read-modify-write needs to be a release; otherwise, the write in the read-modify-write can be any special write. Similarly, to make a write appear atomic with RCpc, read-modify-write operations can be used to replace the appropriate operations analogous to the PC model. As mentioned earlier, writes may also appear non-atomic in more complex implementations of RCsc. Preserving the atomicity of a write can be achieved by labeling sufficient operations as special; however, explaining how this can be done precisely is difficult within the simple framework presented in this article. We should note that the RCsc model is also accompanied by a higher level abstraction (described in Section 7) that relieves the need for the programmer to directly reason with the lower level specification for a large class of programs [13].

6.4.3 Alpha, RMO, and PowerPC

The Alpha, RMO, and PowerPC models all provide explicit fence instructions as their safety nets.

The Alpha model provides two different fence instructions, the memory barrier (MB) and the write memory barrier (WMB). The MB instruction can be used to maintain program order from any memory operations before the MB to any memory operations after the MB. The WMB instruction provides this guarantee only among write operations. The Alpha model does not require a safety net for write atomicity.

The SPARC V9 RMO model provides more flavors of fence instructions. Effectively, a MEMBAR instruction can be customized to order a combination of previous read and write operations with respect to future read and write operations; a four bit encoding is used to specify any combination of read to read, read to write, write to read, and write to write orderings. The fact that a MEMBAR can be used to order a write with respect to a following read alleviates the need for using read-modify-writes to achieve this order, as is required in the SPARC V8 TSO or PSO models. Similar to TSO and PSO, the RMO model does not require a safety net for write atomicity.

The PowerPC model provides a single fence instruction, called the SYNC instruction. For imposing program order, the SYNC instruction behaves similar to the MB instruction of the Alpha model with one exception. The exception is that even if a SYNC is placed between two reads to the same location, it is possible for the second read to return the value of an older write than the first read; i.e., the reads appear to occur out of program order. This can create subtle correctness problems in programs, and may require the use of read-modify-write operations (analogous to their use for PC and RCpc) to enforce program order between two reads to the same location. PowerPC also differs from Alpha and RMO in terms of atomicity in that it allows a write to be seen early by another processor's read; therefore, analogous to PC and RCpc, read-modify-write operations may need to be used to make a write appear atomic.

6.4.4 Compiler Optimizations

Unlike the models in the previous sections, the models that relax all program orders provide sufficient flexibility to allow common compiler optimizations on shared memory operations. In models such as WO, RCsc and RCpc, the compiler has the flexibility to reorder memory operations between two consecutive synchronization or special operations. Similarly, in the Alpha, RMO, and PowerPC models, the compiler has full flexibility to reorder operations between consecutive fence instructions. Since most programs use these operations or instructions infrequently, the compiler gets large regions of code where virtually all optimizations that are used for uniprocessor programs can be safely applied.

7 An Alternate Abstraction for Relaxed Memory Models

The flexibility provided by the relaxed memory models described in the previous section enables a wide range of performance optimizations that have been shown to improve performance substantially [9, 11, 6]. However, the higher performance is accompanied by a higher level of complexity for programmers. Furthermore, the wide range of models supported by different systems requires programmers to deal with various semantics that differ in subtle ways and complicates the task of porting programs across these systems. The programming complexity arises due to the *system-centric* specifications that are typically provided by relaxed memory models. Such specifications directly expose the programmer to the reordering and atomicity optimizations that are allowed by a model, and require the programmer to consider the behavior of the program in the presence of such optimizations in order to reason about its correctness. This provides an incentive to devise a higher level abstraction for programmers that provides a simpler view of the system, and yet allows system designers to exploit the same types of optimizations.

For the relaxed models we have described, the programmer can ensure correctness for a program by using sufficient safety nets (e.g., fence instructions, more conservative operation types, or read-modify-write operations) to impose the appropriate ordering and atomicity requirements on memory operations. The difficult problem is identifying the ordering constraints that are necessary for correctness. For example, consider the program in

Figure 1 executing on a model such as weak ordering (WO). In this example, it is sufficient to maintain only the following orders for correctness: (1) on P1, maintain program order between the write to `Head` and operations before the write to `Head`, and (2) on other processors, maintain the program order from the read of `Head` to the following operations. The write and read of `Head` actually behave as synchronization operations, and by identifying them as such, the appropriate program orders will be automatically maintained by a model like WO. Recognizing this issue, many models such as WO are accompanied by informal conditions for what programmers must do to ensure "correct" behavior. For example, weak ordering requires that programmers should identify all synchronization operations. However, the informal nature of these conditions makes them ambiguous when they are applied over a wide range of programs (e.g., which operations should really be identified as synchronization). Therefore, in a lot of cases, the programmer must still resort to reasoning with low level reordering optimizations to determine whether sufficient orders are enforced.

Instead of exposing performance-enhancing optimizations directly to the programmer as is done by a system-centric specification, a *programmer-centric* specification requires the programmer to provide certain information about the program. This information is then used by the system to determine whether a certain optimization can be applied without violating the correctness of the program. To provide a formal programmer-centric specification, we need to *first* define the notion of "correctness" for programs. An obvious choice for this is sequential consistency since it is a natural extension of the uniprocessor notion of correctness and the most commonly assumed notion of correctness for multiprocessors. *Second*, the information required from the programmer must be defined precisely.

In summary, with the programmer-centric approach, a memory consistency model is described in terms of program-level information that must be provided by the programmer. Systems based on the model exploit the information to perform optimizations without violating sequential consistency. Our previous work has explored various programmer-centric approaches. For example, the *data-race-free-0* (DRF0) approach explores the information that is required to allow optimizations similar to those enabled by weak ordering [2]. The *properly-labeled* (PL) approach was provided along with the definition of release consistency (RCsc) as a simpler way to reason about the type of optimizations exploited by RCsc [13]. Programmer-centric approaches for exploiting more aggressive optimizations are described in our other work [7, 3, 1, 6]; a unified framework for designing programmer-centric models has also been developed and used to explore the design space of such models [1].

To illustrate the programmer-centric approach more concretely, the next section describes the type of program-level information that may be provided by the programmer to enable optimizations similar to those exploited by the weak ordering model. We then describe how such information can actually be conveyed by the programmer to the system.

7.1 An Example Programmer-Centric Framework

Recall that weak ordering is based on the intuition that memory operations can be classified as data and synchronization, and data operations can be executed more aggressively than synchronization operations. A key goal of the programmer-centric approach is to formally define the operations that should be distinguished as synchronization.

An operation must be defined as a synchronization operation if it forms a *race* with another operation in any sequentially consistent execution; other operations can be defined as data. Given a sequentially consistent execution, an operation forms a *race* with another operation if the two operations access the same location, at least one of the operations is a write, and there are no other intervening operations between the two operations under consideration. Consider the example in Figure 12 (same as the example in Figure 5(b)). In every sequentially consistent execution of this program, the write and read of `Data` will always be separated by the intervening operations of the write and read of `Head`. Therefore, the operations on `Data` are data operations. However, the operations on `Head` are not always separated by other operations; therefore, they are synchronization operations. Note that the programmer only reasons about sequentially consistent executions of the program and does not deal with any reordering optimizations in order to provide the above information.

From the system design viewpoint, operations distinguished as synchronization need to be executed conservatively, while operations distinguished as data can be executed aggressively. In particular, the optimizations enabled by the weak ordering model can be safely applied. Furthermore, the information also enables more aggressive optimizations than exploited by weak ordering [2, 13, 1].

As shown in Figure 13, the programmer-centric framework requires the programmer to identify all operations that may be involved in a race as synchronization operations. Other operations may be distinguished as either data

```
    Initially all locations = 0

P1          P2

Data = 2000  while (Head == 0) {;}
Head = 1     ...  = Data
```

Figure 12: Providing information about memory operations.

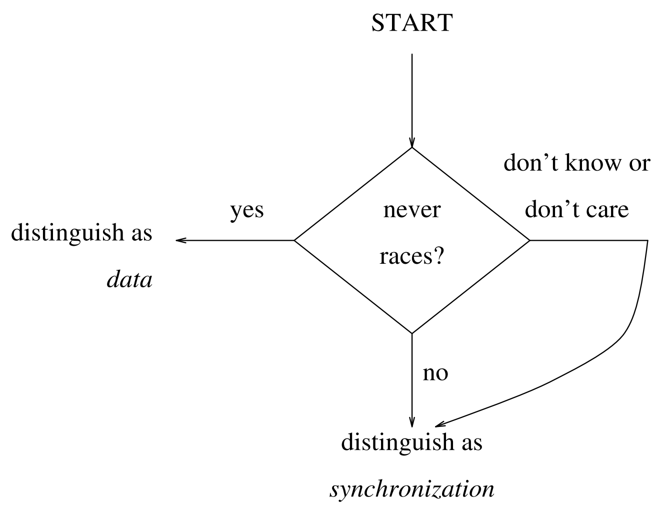


Figure 13: Deciding how to distinguish a memory operation.

or synchronization. Therefore, an operation may be conservatively distinguished as a synchronization operation if the programmer is not sure whether the particular operation is involved in a race or not. This "don't-know" option is important for the following reasons. A programmer can trivially ensure correctness by conservatively distinguishing all operations as synchronization; of course, this forgoes any performance gains but potentially allows a faster path to an initial working program. Another potential benefit of the don't-know option is that it allows the programmer to incrementally tune performance by providing accurate information for a subset of the memory operations (in performance-critical areas of the program), and simply providing conservative information for the remaining operations. Of course, correctness is not guaranteed if the programmer incorrectly distinguishes a race operation as data.

Providing the appropriate information to the system requires a mechanism at the programming language level to distinguish memory operations, and also a mechanism for passing this information in some form to the hardware level. We describe such mechanisms in the next section.

7.2 Mechanisms for Distinguishing Memory Operations

This section describes several possible mechanisms for conveying the information required by the programmer-centric framework described in the previous section.

7.2.1 Conveying Information at the Programming Language Level

We consider programming languages with explicit parallel constructs. The parallel programming support provided by the language may range from high level parallelism constructs such as doall loops to low level use of memory operations for achieving synchronization. Therefore, the mechanism for conveying information about memory operations depends on the support for parallelism provided by the language.

Many languages specify high level paradigms for parallel tasks and synchronization, and restrict programmers to using these paradigms. For example, consider a language that only allows parallelism to be expressed through doall loops. Correct use of doall loops implies that no two parallel iterations of the loop should access the same location if at least one of the accesses is a write. Thus, the information about memory operations is implicitly conveyed since none of the operations in the high level program are involved in a race.

At a slightly lower level, the language may provide a library of common synchronization routines, and the programmer is restricted to achieve synchronization by calls to such routines. In this case, the programmer must use sufficient such synchronization calls to eliminate any races between other operations in the program. Therefore, similar to the case with doall loops, the information about all memory operations visible to the programmer (i.e., excluding operations used within the synchronization routines) is implicitly conveyed. Of course, the compiler or writers of library routines must still ensure that the operation types (i.e., synchronization or data) for additional operations introduced to implement constructs such as doall loops or other synchronization routines are conveyed to the lower levels such as the hardware.

Finally, the programmer may be allowed to directly use memory operations visible at the program level for synchronization purposes (e.g., using a memory location as a flag variable). In this case, the programmer must explicitly convey information about operation types. One way to do this is to associate this information with the static instructions at the program level. For example, the language may provide constructs that identify specific static regions of code to be synchronization (or data); then all dynamic operations generated from that region of code are implicitly identified as synchronization (or data). Another option is to associate the data or synchronization attribute with a shared variable or address. For example, the language may provide additional type declarations that allow the programmer to identify variables that are used for synchronization purposes.

The type and generality of the mechanisms provided by the programming language affects the ease of use for conveying the required information. For example, in the method where type declarations are used to indicate the operation type, a default where all operations are considered data (unless indicated otherwise) can be beneficial since data operations are more frequent. On the other hand, making the synchronization type the default makes it simpler to bring up an initial working program, and can potentially decrease errors by requiring programmers to explicitly declare the more aggressive data operations.

7.2.2 Conveying Information to the Hardware

The information conveyed at the programming language level must ultimately be provided to the underlying hardware. Therefore, the compiler is often responsible for appropriately translating the higher level information to a form that is supported by the hardware.

Similar to the mechanisms used at the programming language level, information about memory operations may be associated with either specific address ranges or with the memory instruction corresponding to the operation. One way to associate the information with specific address ranges is to treat operations to specific pages as data or synchronization operations. Associating the information with a specific memory instruction can be done in one of two ways. The first option is to provide multiple flavors of memory instructions (e.g., by providing extra opcodes) to distinguish memory operations. The second option is to use any unused high order bits of the virtual memory address to achieve this (i.e., address shadowing). Finally, some memory instructions, such as compare-and-swap or load-locked/store-conditional, may be treated as synchronization by default.

Most commercial systems do not provide the above functionality for directly communicating information about memory operations to the hardware. Instead, this information must be transformed to explicit fence instructions supported at the hardware level to impose sufficient ordering constraints. For example, to provide the semantics of synchronization operations of weak ordering on hardware that supports Alpha-like memory barriers, the compiler can precede and follow every synchronization operation with a memory barrier.

8 Discussion

There is strong evidence that relaxed memory consistency models provide better performance than is possible with sequential consistency by enabling a number of hardware optimizations [9, 11, 6]. The increase in processor speeds relative to memory and communication speeds will only increase the potential benefit from these models. In addition to providing performance gains at the hardware level, relaxed memory consistency models also play a key role in enabling important compiler optimizations. The above reasons have led many commercial architectures, such as Digital Alpha, Sun SPARC, and IBM PowerPC, to support relaxed memory models. Furthermore, nearly all other architectures also support some form of explicit fence instructions that indicates a commitment to support relaxed memory models in the future. Unfortunately, the existing literature on memory consistency models is vast and complex, with most of it targeted towards researchers in this area rather than typical users or builders of computer systems. This article used a uniform and intuitive terminology to cover several issues related to memory consistency models representative of those used in industry today, with the goal of reaching the wider community of computer professionals.

One disadvantage of relaxed memory consistency models is the increase in programming complexity. Much of this complexity arises because many of the specifications presented in the literature expose the programmer to low level performance optimizations that are enabled by the model. Our previous work has addressed this issue by defining models using a higher level abstraction; this abstraction provides the illusion of sequential consistency as long as the programmer provides correct program-level information about memory operations. Meanwhile, language standardization efforts such as High Performance Fortran have led to high-level memory models that are different from sequential consistency. For example, the forall statement of High Performance Fortran, which specifies a computation for a set of array indices, has a copy-in/copy-out semantics, where the computation for one index is not affected by values produced by the computation of other indices. Overall, the choice of the best memory consistency model is far from resolved and would benefit from more active collaboration between language and hardware designers.

9 Acknowledgements

Much of this work was done as part of our dissertation research. We are indebted to our respective advisors, Mark Hill, and Anoop Gupta and John Hennessy, for their direction throughout our dissertation work. We especially thank Mark Hill for suggesting the need for this paper and for encouraging us to write it. We thank Sandhya Dwarkadas, Anoop Gupta, John Hennessy, Mark Hill, Yuan Yu, and Willy Zwaenepoel for their valuable comments on earlier versions of this paper. We also thank Andreas Nowatzyk, Steve Scott, and Wolf-Dietrich Weber for information on products developed by Sun Microsystems, Cray Research, and HaL Computer Systems, respectively.

References

[1] Sarita V. Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Computer Sciences Department, University of Wisconsin-Madison, December 1993. Available as Technical Report #1198.

[2] Sarita V. Adve and Mark D. Hill. Weak ordering - A new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[3] Sarita V. Adve and Mark D. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[4] Francisco Corella, Janice M. Stone, and Charles M. Barton. A formal specification of the PowerPC shared memory architecture. Technical Report Computer Science Technical Report RC 18638(81566), IBM Research Division, T.J. Watson Research Center, January 1993.

[5] Michel Dubois, Christoph Scheurich, and Fayé Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, June 1986.

[6] Kourosh Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Stanford University, 1995.

[7] Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John L. Hennessy, and Mark D. Hill. Programming for different memory consistency models. *Journal of Parallel and Distributed Computing*, 15(4):399–407, August 1992.

[8] Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John L. Hennessy, and Mark D. Hill. Specifying system requirements for memory consistency models. Technical Report CSL-TR-93-594, Stanford University, December 1993. Also available as Computer Sciences Technical Report #1199, University of Wisconsin - Madison.

[9] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–257, April 1991.

[10] 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*, pages I:355–364, August 1991.

[11] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Hiding memory latency using dynamic scheduling in shared-memory multiprocessors. In *Proceeding of the 19th Annual International Symposium on Computer Architecture*, pages 22–33, May 1992.

[12] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Revision to "Memory consistency and event ordering in scalable shared-memory multiprocessors". Technical Report CSL-TR-93-568, Stanford University, April 1993.

[13] Kourosh Gharachorloo, Dan Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[14] *IBM System/370 Principles of Operation*. IBM, May 1983. Publication Number GA22-7000-9, File Number S370-01.

[15] Arvind Krishnamurthy and Katherine Yelick. Optimizing parallel SPMD programs. In *Languages and Compilers for Parallel Computing*, 1994.

[16] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[17] Cathy May, Ed Silha, Rick Simpson, and Hank Warren, editors. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann Publishers, Inc., 1994.

[18] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[19] Richard L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, 1992.

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

[21] David L. Weaver and Tom Germond, editors. *The SPARC Architecture Manual*. Prentice Hall, 1994. SPARC International, Version 9.