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]This work has been submitted to the IEEE for possible publication. Copyright may be transferred without notice, after which this version will be superseded.

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.

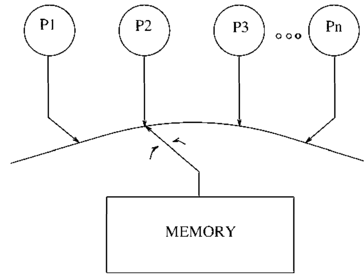


Figure 3: Programmer's view of sequential consistency.

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

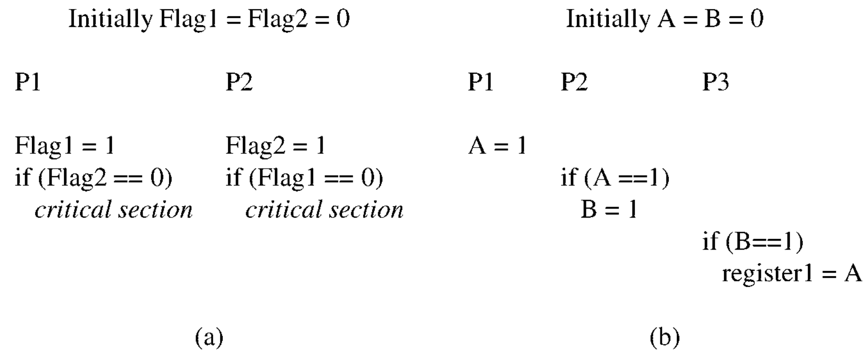


Figure 4: Examples for sequential consistency.

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

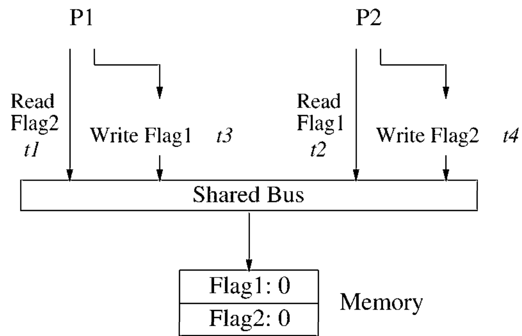
We have chosen three canonical hardware optimizations as illustrative examples of typical interactions that arise in implementing sequential consistency in the absence of data caching. A large number of other common hardware optimizations can lead to interactions similar to those illustrated by our canonical 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.

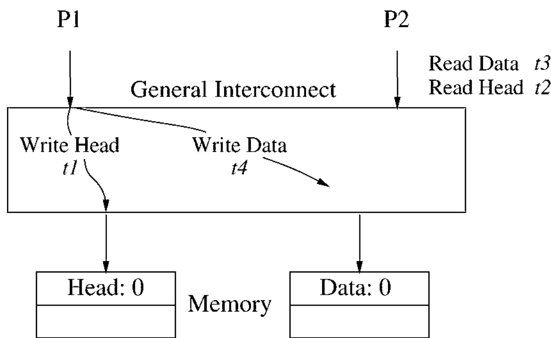
To see how the use of write buffers can violate sequential consistency, consider the program in Figure 5(a). The program depicts Dekker's algorithm also shown earlier in Figure 4(a). As explained earlier, a sequentially consistent system must prohibit an outcome where both the reads of the flags return 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.



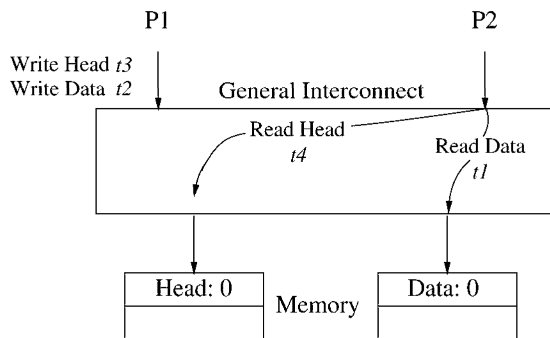
P1	P2
Flag1 = 1	Flag2 = 1
if (Flag2 == 0)	if (Flag1 == 0)
critical section	*critical section*

(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

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.