CHAPTER 9 Hardware-Software Tradeoffs

9.1 Introduction

This chapter focuses on addressing some potential limitations of the directory-based, cache-coherent systems discussed in the previous chapter, and the hardware-software tradeoffs that this engenders. The primary limitations of those systems, in both performance and cost, are the following.

- *High waiting time at memory operations*. Sequential consistency (SC) was assumed to be the memory consistency model of choice, and the chapter discussed how systems might satisfy the sufficient conditions for SC. To do this, a processor would have to wait for the previous memory operation to complete before issuing the next one. This has even greater impact on performance in scalable systems than in bus-based systems, since communication latencies are longer and there are more network transactions in the critical path. Worse still, it is very limiting for compilers, which cannot reorder memory operations to shared data at all if the programmer assumes sequential consistency.

- *Limited capacity for replication*. Communicated data are automatically replicated only in the processor cache, not in local main memory. This can lead to capacity misses and artifactual communication when working sets are large and include nonlocal data or when there are a lot of conflict misses.
- *High design and implementation cost*. The communication assist contains hardware that is specialized for supporting cache-coherence and tightly integrated into the processing node. Protocols are also complex, and getting them right in hardware takes substantial design time. By cost here we mean the cost of hardware and of system design time. Recall from Chapter 3 that there is also a programming cost associated with achieving good performance, and approaches that reduce system cost can often increase this cost dramatically.

The chapter will focus on these three limitations, the first two of which have primarily to do with performance and the third primarily with cost. The approaches that have been developed to address them are still controversial to varying extents, but aspects of them are finding their way into mainstream parallel architecture. There are other limitations as well, including the addressability limitations of a shared physical address space—as discussed for the Cray T3D in Chapter 7—and the fact that a single protocol is hard-wired into the machine. However, solutions to these are often incorporated in solutions to the others, and they will be discussed as advanced topics.

The problem of waiting too long at memory operations can be addressed in two ways in hardware. First, the implementation can be designed to not satisfy the sufficient conditions for SC, which modern non-blocking processors are not inclined to do anyway, but to satisfy the SC model itself. That is, a processor needn't wait for the previous operation to complete before issuing the next one; however, the system ensures that operations do not complete or become visible to other processors out of program order. Second, the memory consistency model can itself be relaxed so program orders do not have to be maintained so strictly. Relaxing the consistency model changes the semantics of the shared address space, and has implications for both hardware and software. It requires more care from the programmer in writing correct programs, but enables the hardware to overlap and reorder operations further. It also allows the compiler to reorder memory operations within a process before they are even presented to hardware, as optimizing compilers are wont to do. Relaxed memory consistency models will be discussed in Section 9.2.

The problem of limited capacity for replication can be addressed by automatically caching data in main memory as well, not just in the processor caches, and by keeping these data coherent. Unlike in hardware caches, replication and coherence in main memory can be performed at a variety of granularities—for example a cache block, a page, or a user-defined object—and can be managed either directly by hardware or through software. This provides a very rich space of protocols, implementations and cost-performance tradeoffs. An approach directed primarily at improving performance is to manage the local main memory as a hardware cache, providing replication and coherence at cache block granularity there as well. This approach is called cache-only memory architecture or COMA, and will be discussed in Section 9.3. It relieves software from worrying about capacity misses and initial distribution of data across main memories, while still providing coherence at fine granularity and hence avoiding false-sharing. However, it is hardware-intensive, and requires per-block tags to be maintained in main memory as well.

Finally, there are many approaches to reduce hardware cost. One approach is to integrate the communication assist and network less tightly into the processing node, increasing communication latency and occupancy. Another is to provide automatic replication and coherence in soft-

ware rather than hardware. The latter approach provides replication and coherence in main memory, and can operate at a variety of granularities. It enables the use of off-the-shelf commodity parts for the nodes and interconnect, reducing hardware cost but pushing much more of the burden of achieving good performance on to the programmer. These approaches to reduce hardware cost are discussed in Section 9.4.

The three issues are closely related. For example, cost is strongly related to the manner in which replication and coherence are managed in main memory: The coarser the granularities used, the more likely that the protocol can be synthesized through software layers, including the operating system, thus reducing hardware costs (see Figure 9-1). Cost and granularity are also related to the

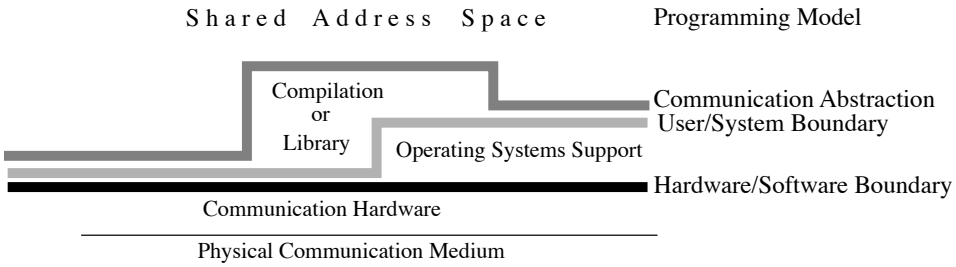

Figure 9-1 Layers of the communication architecture for systems discussed in this chapter.

memory consistency model: Lower-cost solutions and larger granularities benefit more from relaxing the memory consistency model, and implementing the protocol in software makes it easier to fully exploit the relaxation of the semantics. To summarize and relate the alternatives, Section 9.5 constructs a framework for the different types of systems based on the granularities at which they allocate data in the replication store, keep data coherent and communicate data. This leads naturally to an approach that strives to achieve a good compromise between the high-cost COMA approach and the low-cost all-software approach. This approach, called Simple-COMA, is discussed in Section 9.5 as well.

The implications of the systems discussed in this chapter for parallel software, beyond those discussed in the previous chapters, are explored in Section 9.6. Finally, Section 9.7 covers some advanced topics, including the techniques to address the limitations of a shared physical address space and a fixed coherence protocol.

9.2 Relaxed Memory Consistency Models

Recall from Chapter 5 that the memory consistency model for a shared address space specifies the constraints on the order in which memory operations (to the same or different locations) can appear to execute with respect to one another, enabling programmers to reason about the behavior and correctness of their programs. In fact, any system layer that supports a shared address space naming model has a memory consistency model: the programming model or programmer's interface, the communication abstraction or user-system interface, and the hardware-software interface. Software that interacts with that layer must be aware of its memory consistency model. We shall focus mostly on the consistency model as seen by the programmer—i.e. at the interface

between the programmer and the rest of the system composed of the compiler, OS and hardware—since that is the one with which programmers reason[1]. For example, a processor may preserve all program orders presented to it among memory operations, but if the compiler has already reordered operations then programmers can no longer reason with the simple model exported by the hardware.

Other than programmers (including system programmers), the consistency model at the programmer's interface has implications for programming languages, compilers and hardware as well. To the compiler and hardware, it indicates the constraints within which they can reorder accesses from a process and the orders that they cannot appear to violate, thus telling them what tricks they can play. Programming languages must provide mechanisms to introduce constraints if necessary, as we shall see. In general, the fewer the reorderings of memory accesses from a process we allow the system to perform, the more intuitive a programming model we provide to the programmer, but the more we constrain performance optimizations. The goal of a memory consistency model is to impose ordering constraints that strike a good balance between programming complexity and performance. The model should also be portable; that is, the specification should be implementable on many platforms, so that the same program can run on all these platforms and preserve the same semantics.

The sequential consistency model that we have assumed so far provides an intuitive semantics to the programmer—program order within each process and a consistent interleaving across processes—and can be implemented by satisfying its sufficient conditions. However, its drawback is that it by preserving a strict order among accesses it restricts many of the performance optimizations that modern uniprocessor compilers and microprocessors employ. With the high cost of memory access, computer systems achieve higher performance by reordering or overlapping the servicing of multiple memory or communication operations from a processor. Preserving the sufficient conditions for SC clearly does not allow for much reordering or overlap. With SC at the programmer's interface, the compiler cannot reorder memory accesses even if they are to different locations, thus disallowing critical performance optimizations such as code motion, common-subexpression elimination, software pipelining and even register allocation.

Example 9-1 Show how register allocation can lead to a violation of SC even if the hardware satisfies SC.

Answer Consider the code fragment shown in Figure 9-2(a). After register allocation, the code produced by the compiler and seen by hardware might look like that in Figure 9-2(b). The result (u,v) = (0,0) is disallowed under SC hardware in (a), but not only can but will be produced by SC hardware in (b). In effect, register allocation reorders the write of A and the read of B on P1 and reorders the write of B and the read of A on P2. A uniprocessor compiler might easily perform these

1. The term programmer here refers to the entity that is responsible for generating the parallel program. If the human programmer writes a sequential program that is automatically parallelized by system software, then it is the system software that has to deal with the memory consistency model; the programmer simply assumes sequential semantics as on a uniprocessor.

optimizations in each process: They are valid for sequential programs since the reordered accesses are to different locations.

P1	P2
B = 0	A = 0
A = 1	B = 1
u = B	v = A

(a) Before register allocation

P1	P2
r1 = 0	r2 = 0
A = 1	B = 1
u = r1	v = r2
B = r1	A = r2

(a) After register allocation

Figure 9-2 Example showing how register allocation by the compiler can violate SC.

The code in (a) is the original code with which the programmer reasons under SC. r1, r2 are registers.

SC at the programmer's interface implies SC at the hardware-software interface; if the sufficient conditions for SC are met, a processor waits for an access to complete or at least commit before issuing the next one, so most of the latency suffered by memory references is directly seen by processors as stall time. While a processor may continue executing non-memory instructions while a single outstanding memory reference is being serviced, the expected benefit from such overlap is tiny (even without instruction-level parallelism on average every third instruction is a memory reference [HeP95]). We need to do something about this performance problem.

One approach we can take is to preserve sequential consistency at the programmer's interface but hide the long latency stalls from the processor. We can do this in several ways, divided into two categories [GGH91]. The techniques and their performance implications will be discussed further in Chapter 11; here, we simply provide a flavor for them. In the first category, the system still preserves the sufficient conditions for SC. The compiler does not reorder memory operations. Latency tolerance techniques such as prefetching of data or multithreading (see Chapter 11) are used to overlap accesses with one another or with computation—thus hiding much of their latency from the processor—but the actual read and write operations are not issued before previous ones complete and the operations do not become visible out of program order.

In the second category, the system preserves SC but not the sufficient conditions at the programmer's interface. The compiler can reorder operations as long as it can guarantee that sequential consistency will not be violated in the results. Compiler algorithms have been developed for this [ShS88,KrY94], but they are expensive and their analysis is currently quite conservative. At the hardware level, memory operations are issued and executed out of program order, but are guaranteed to become visible to other processors in program order. This approach is well suited to dynamically scheduled processors that use an instruction lookahead buffer to find independent instructions to issue. The instructions are inserted in the lookahead buffer in program order, they are chosen from the instruction lookahead buffer and executed out of order, but they are guaranteed to retire from the lookahead buffer in program order. Operations may even issue and execute out of order past an unresolved branch in the lookahead buffer based on branch prediction—called speculative execution—but since the branch will be resolved and retire before them they will not become visible to the register file or external memory system before the branch is resolved. If the branch was mis-predicted, the effects of those operations will never become visi-

ble. The technique called *speculative reads* goes a little further; here, the values returned by reads are used even before they are known to be correct; later checks determine if they were incorrect, and if so the computation is rolled back to reissue the read. Note that it is not possible to speculate with stores in this manner because once a store is made visible to other processors it is extremely difficult to roll back and recover: a store's value should not be made visible to other processors or the external memory system environment until all previous references have correctly completed.

Some or all of these techniques are supported by many modern microprocessors, such as the MIPS R10000, the HP PA-8000, and the Intel Pentium Pro. However, they do require substantial hardware resources and complexity, their success at hiding multiprocessor latencies is not yet clear (see Chapter 11), and not all processors support them. Also, these techniques work for processors, but they do not help compilers perform the reorderings of memory operations that are critical for their optimizations.

A completely different way to overcome the performance limitations imposed by SC is to change the memory consistency model itself; that is, to not guarantee such strong ordering constraints to the programmer, but still retain semantics that are intuitive enough to a programmer. By relaxing the ordering constraints, these *relaxed consistency models* allow the compiler to reorder accesses before presenting them to the hardware, at least to some extent. At the hardware level, they allow multiple memory accesses from the same process to be outstanding at a time and even to complete or become visible out of order, thus allowing much of the latency to be overlapped and hidden from the processor. The intuition behind relaxed models is that SC is usually too conservative, in that many of the orders it preserves are not really needed to satisfy a programmer's intuition in most situations. Much more detailed treatments of relaxed memory consistency models can be found in [Adv93,Gha95].

Consider the simple example shown in Figure 9-3. On the left are the orderings that will be main-

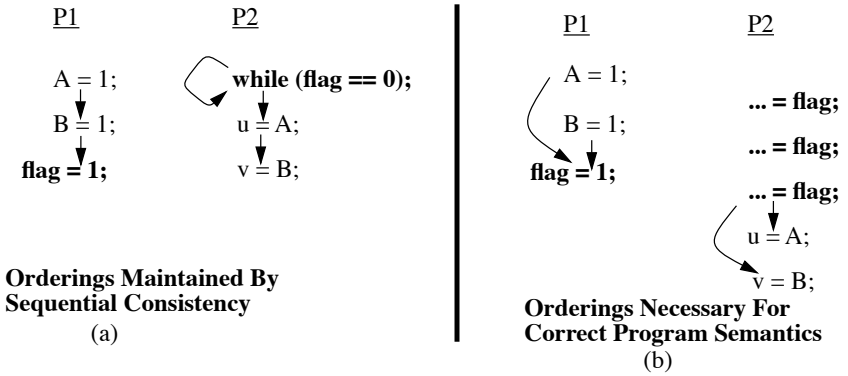

Figure 9-3 Intuition behind relaxed memory consistency models.

The arrows in the figure indicate the orderings maintained. The left side of the figure (part (a)) shows the orderings maintained by the sequential consistency model. The right side of the figure (part (b)) shows the orderings that are *necessary* for "correct / intuitive" semantics.

tained by an SC implementation. On the right are the orderings that are necessary for intuitively correct program semantics. The latter are far fewer. For example, writes to variables `A` and `B` by P1 can be reordered without affecting the results observed by the program; all we must ensure is

that both of them complete before variable `flag` is set to 1. Similarly, reads to variables `A` and `B` can be reordered at P2, once `flag` has been observed to change to value 1.[1] Even with these reorderings, the results look just like those of an SC execution. On the other hand, while the accesses to `flag` are also simple variable accesses, a model that allowed them to be reordered with respect to `A` and `B` at either process would compromise the intuitive semantics and SC results. It would be wonderful if system software or hardware could automatically detect which program orders are critical to maintaining SC semantics, and allow the others to be violated for higher performance [ShS88]. However, the problem is undecidable for general programs, and inexact solutions are often too conservative to be very useful.

There are three parts to a complete solution for a relaxed consistency model: the system specification, the programmer's interface, and a translation mechanism.

The system specification. This is a clear specification of two things. First, what program orders among memory operations are guaranteed to be preserved, in an observable sense, by the system, including whether write atomicity will be maintained. And second, if not all program orders are preserved by default, then what mechanisms does the system provide for a programmer to enforce those orderings explicitly. As should be clear by now, each of the compiler and the hardware has a system specification, but we focus on the specification that the two together or the system as a whole presents to the programmer. For a processor architecture, the specification it exports governs the reorderings that it allows and the order-preserving primitives it provides, and is often called the processor's memory model.

The programmer's interface. The system specification is itself a consistency model. A programmer may use it to reason about correctness and insert the appropriate order-preserving mechanisms. However, this is a very low level interface for a programmer: Parallel programming is difficult enough without having to think about reorderings and write atomicity! The specific reorderings and order-enforcing mechanisms supported are different across systems, compromising portability. Ideally, a programmer would assume SC semantics, and the system components would automatically determine what program orders they may alter without violating this illusion. However, this is too difficult to compute. What a programmer therefore wants is a methodology for writing "safe" programs. This is a contract such that if the program follows certain high-level rules or provides enough program annotations—such as telling the system that `flag` in Figure 9-3 is in fact used as a synchronization variable—then any system on which the program runs will always guarantee a sequentially consistent execution, regardless of the default reorderings in the system specifications it supports. The programmer's responsibility is to follow the rules and provide the annotations (which hopefully does not involve reasoning at the level of potential reorderings); the system's responsibility is to maintain the illusion of sequential consistency. Typically, this means using the annotations as constraints on compiler reorderings, and also translating the annotations to the right order-preserving mechanisms for the processor at the right places. The implication for programming languages is that they should support the necessary annotations.

The translation mechanism. This translates the programmer's annotations to the interface exported by the system specification.

1. Actually, it is possible to further weaken the requirements for correct execution. For example, it is not necessary for stores to A and B to complete before store to Flag is done; it is only necessary that they be complete by the time processor P2 observes that the value of Flag has changed to 1. It turns out, such relaxed models are extremely difficult to implement in hardware, so we do not discuss them here. In software, some of these relaxed models make sense, and we will discuss them in Section 9.3.

We organize our discussion of relaxed consistency models around these three pieces. We first examine different low-level specifications exported by systems and particularly microprocessors, organizing them by the types of reorderings they allow. Section 9.2.2 discusses the programmer's interface or contract and how the programmer might provide the necessary annotations, and Section 9.2.3 briefly discusses translation mechanisms. A detailed treatment of implementation complexity and performance benefits will be postponed until we discuss latency tolerance mechanisms in Chapter 11.

9.2.1 System Specifications

Several different reordering specifications have been proposed by microprocessor vendors and by researchers, each with its own mechanisms for enforcing orders. These include total store ordering (TSO) [SFC91, Spa91], partial store ordering (PSO) [SFC91, Spa91], and relaxed memory ordering (RMO) [WeG94] from the SUN Sparc V8 and V9 specifications, processor consistency described in [Goo89, GLL+90] and used in the Intel Pentium processors, weak ordering (WO) [DSB86,DuS90], release consistency (RC) [GLL+90], and the Digital Alpha [Sit92] and IBM/Motorola PowerPC [MSS+94] models. Of course, a particular implementation of a processor may not support all the reorderings that its system specification allows. The system specification defines the semantic interface for that architecture, i.e. what orderings the programmer must assume might happen; the implementation determines what reorderings actually happen and how much performance can actually be gained.

Let us discuss some of the specifications or consistency models, using the relaxations in program order that they allow as our primary axis for grouping models together [Gha95]. The first set of models only allows a read to bypass (complete before) an earlier incomplete write in program order; i.e. allows the write->read order to be reordered (TSO, PC). The next set allows this and also allows writes to bypass previous writes; i.e. write->write reordering (PSO). The final set allows reads or writes to bypass previous reads as well; that is, allows all reorderings among read and write accesses (WO, RC, RMO, Alpha, PowerPC). Note that a read-modify-write operation is treated as being both a read and a write, so it is reordered with respect to another operation only if both a read and a write can be reordered with respect to that operation. Of course, in all cases we assume basic cache coherence—write propagation and write serialization—and that uniprocessor data and control dependences are maintained within each process. The specifications discussed have in most cases been motivated by and defined for the processor architectures themselves, i.e. the hardware interface. They are applicable to compilers as well, but since sophisticated compiler optimizations require the ability to reorder all types of accesses most compilers have not supported as wide a variety of ordering models. In fact, at the programmer's interface, all but the last set of models have their utility limited by the fact that they do not allow many important compiler optimizations. The focus in this discussion is mostly on the models; implementation issues and performance benefits will be discussed in Chapter 11.

Relaxing the *Write-to-Read* Program Order

The main motivation for this class of models is to allow the hardware to hide the latency of write operations that miss in the first-level cache. While the write miss is still in the write buffer and not yet visible to other processors, the processor can issue and complete reads that hit in its cache or even a single read that misses in its cache. The benefits of hiding write latency can be substantial, as we shall see in Chapter 11, and most processors can take advantage of this relaxation.

How well do these models preserve the programmer's intuition even without any special operations? The answer is quite well, for the most part. For example, spinning on a flag for event synchronization works without modification (Figure 9-4(a)). The reason is that TSO and PC models preserve the ordering of writes, so the write of the flag is not visible until all previous writes have completed in the system. For this reason, most early multiprocessors supported one of these two models, including the Sequent Balance, Encore Multimax, VAX-8800, SparcCenter1000/2000, SGI 4D/240, SGI Challenge, and even the Pentium Pro Quad, and it has been relatively easy to port even complex programs such as the operating systems to these machines.

Of course, the semantics of these models is not SC, so there are situations where the differences show through and SC-based intuition is violated. Figure 9-4 shows four code examples, the first three of which we have seen earlier, where we assume that all variables start out having the value 0. In segment (b), SC guarantees that if B is printed as 1 then A too will be printed as 1, since the

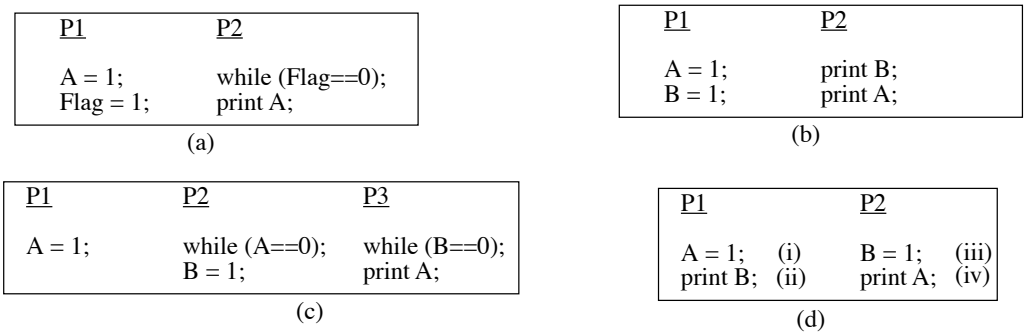

Figure 9-4 Example code sequences repeated to compare TSO, PC and SC.

Both TSO and PC provide same results as SC for code segments (a) and (b), PC can violate SC semantics for segment (c) (TSO still provides SC semantics), and both TSO and PC violate SC semantics for segment (d).

writes of A and B by P1 cannot be reordered. For the same reason, TSO and PC also have the same semantics. For segment (c), only TSO offers SC semantics and prevents A from being printed as 0, not PC. The reason is that PC does not guarantee write atomicity. Finally, for segment (d) under SC no interleaving of the operations can result in 0 being printed for both A and B. To see why, consider that program order implies the precedence relationships (i) -> (ii) and (iii) -> (iv) in the interleaved total order. If B=0 is observed, it implies (ii) -> (iii), which therefore implies (i) -> (iv). But (i) -> (iv) implies A will be printed as 1. Similarly, a result of A=0 implies B=1. A popular software-only mutual exclusion algorithm called Dekker's algorithm—used in the absence of hardware support for atomic read-modify-write operations [TW97]—relies on the property that both A and B will not be read as 0 in this case. SC provides this property, further contributing to its view as an intuitive consistency model. Neither TSO nor PC guarantees it, since they allow the read operation corresponding to the print to complete before previous writes are visible.

To ensure SC semantics when desired (e.g., to port a program written under SC assumptions to a TSO system), we need mechanisms to enforce two types of extra orderings: (i) to ensure that a read does not complete before an earlier write (applies to both TSO and PC), and (ii) to ensure write atomicity for a read operation (applies only to PC). For the former, different processor architectures provide somewhat different solutions. For example, the SUN Sparc V9 specification

[WeG94] provides *memory barrier* (MEMBAR) or *fence* instructions of different flavors that can ensure any desired ordering. Here, we would insert a write-to-read ordering flavored MEMBAR before the read. This MEMBAR flavor prevents a read that follows it in program order from issuing before a write that precedes it has completed. On architectures that do not have memory barriers, it is possible to achieve this effect by substituting an atomic read-modify-write operation or sequence for the original read. A read-modify-write is treated as being both a read and a write, so it cannot be reordered with respect to previous writes in these models. Of course the value written in the read-modify-write must be the same as the value read to preserve correctness. Replacing a read with a read-modify-write also guarantees write atomicity at that read on machines supporting the PC model. The details of why this works are subtle, and the interested reader can find them in the literature [AGG+93].

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

Allowing writes as well to bypass previous outstanding writes to different locations allows multiple writes to be merged in the write buffer before updating main memory, and thus multiple write misses to be overlapped and become visible out of order. The motivation is to further reduce the impact of write latency on processor stall time, and to improve communication efficiency between processors by making new data values visible to other processors sooner. SUN Sparc's PSO model [SFC91, Spa91] is the only model in this category. Like TSO, it guarantees write atomicity.

Unfortunately, reordering of writes can violate our intuitive SC semantics quite a bit. Even the use of ordinary variables as flags for event synchronization (Figure 9-4(a)) is no longer guaranteed to work: The write of `flag` may become visible to other processors before the write of `A`. This model must therefore demonstrate a substantial performance benefit to be attractive.

The only additional instruction we need over TSO is one that enforces write-to-write ordering in a process's program order. In SUN Sparc V9, this can be achieved by using a MEMBAR instruction with the write-to-write flavor turned on (the earlier Sparc V8 provided a special instruction called store-barrier or STBAR to achieve this effect). For example, to achieve the intuitive semantics we would insert such an instruction between the writes of `A` and `flag`.

Relaxing *All* Program Orders: Weak Ordering and Release Consistency

In this final class of specifications, no program orders are guaranteed by default (other than data and control dependences within a process, of course). The benefit is that multiple read requests can also be outstanding at the same time, can be bypassed by later writes in program order, and can themselves complete out of order, thus allowing us to hide read latency. These models are particularly useful for dynamically scheduled processors, which can in fact proceed past read misses to other memory references. They are also the only ones that allow many of the reorderings (even elimination) of accesses that are done by compiler optimizations. Given the importance of these compiler optimizations for node performance, as well as their transparency to the programmer, these may in fact be the only reasonable high-performance models for multiprocessors unless compiler analysis of potential violations of consistency makes dramatic advances. Prominent models in this group are weak ordering (WO) [DSB86,DuS90], release consistency (RC) [GLL+90], Digital Alpha [Sit92], Sparc V9 relaxed memory ordering (RMO) [WeG94], and IBM PowerPC [MSS+94,CSB93]. WO is the seminal model, RC is an extension of WO supported by the Stanford DASH prototype [LLJ+93], and the last three are supported in commercial

architectures. Let us discuss them individually, and see how they deal with the problem of providing intuitive semantics despite all the reordering; for instance, how they deal with the flag example.

Weak Ordering: The motivation behind the weak ordering model (also known as the weak consistency model) is quite simple. Most parallel programs use synchronization operations to coordinate accesses to data when this is necessary. Between synchronization operations, they do not rely on the order of accesses being preserved. Two examples are shown in Figure 9-5.The left segment uses a lock/unlock pair to delineate a critical section inside which the head of a linked list is updated. The right segment uses flags to control access to variables participating in a producer-consumer interaction (e.g., A and D are produced by P1 and consumed by P2). The key in the flag example is to think of the accesses to the flag variables as synchronization operations, since that is indeed the purpose they are serving. If we do this, then in both situations the intuitive semantics are not violated by any read or write reorderings that happen between synchronization accesses (i.e. the critical section in segment (a) and the four statements after the while loop in segment (b)). Based on these observations, weak ordering relaxes all program orders by default, and guarantees that orderings will be maintained only at synchronization operations that can be identified by the system as such. Further orderings can be enforced by adding synchronization operations or labeling some memory operations as synchronization. How appropriate operations are identified as synchronization will be discussed soon.

```
P1, P2,..., Pn

...
Lock(TaskQ)
newTask->next = Head;
if (Head != NULL)
  Head->prev = newTask;
Head = newTask;
UnLock(TaskQ)
...
```

(a)

```
      P1                          P2

TOP: while(flag2==0);      TOP: while(flag1==0);
     A = 1;                     x = A;
     u = B;                     y = D;
     v = C;                     B = 3;
     D = B * C;                 C = D / B;
     flag2 = 0;                 flag1 = 0;
     flag1 = 1;                 flag2 = 1;
     goto TOP;                  goto TOP;
```

(b)

Figure 9-5 Use of synchronization operations to coordinate access to ordinary shared data variables.

The synchronization may be through use of explicit lock and unlock operations (e.g., implemented using atomic test-&-set instructions) or through use of flag variables for producer-consumer sharing.

The left side of Figure 9-6 illustrates the reorderings allowed by weak ordering. Each block with a set of reads/writes represents a contiguous run of non-synchronization memory operations from a processor. Synchronization operations are shown separately. Sufficient conditions to ensure a WO system are as follows. Before a synchronization operation is issued, the processor waits for *all* previous operations in program order (both reads and writes) to have completed. Similarly, memory accesses that follow the synchronization operation are not issued until the synchronization operation completes. Read, write and read-modify-write operations that are not labeled as synchronization can be arbitrarily reordered between synchronization operations. Especially when synchronization operations are infrequent, as in many parallel programs, WO typically provides considerable reordering freedom to the hardware and compiler.

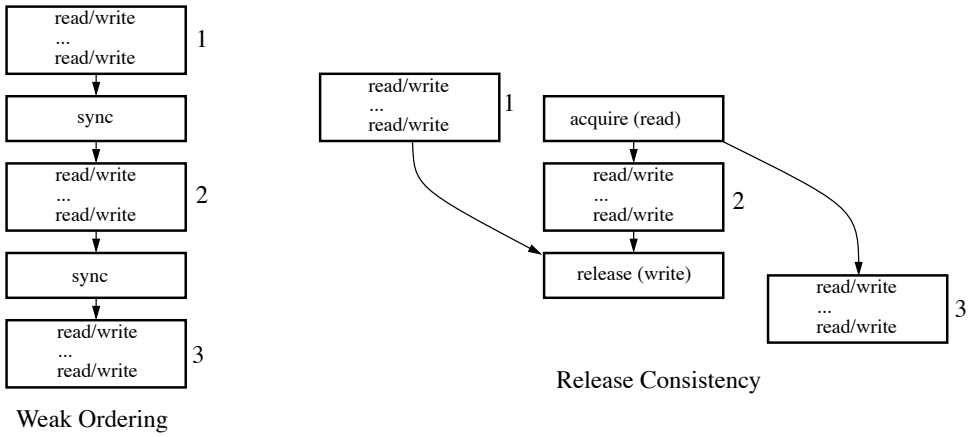

Figure 9-6 Comparison of weak-ordering and release consistency models.

The operations in block 1 precede the first synchronization operation, which is an acquire, in program order. Block 2 is between the two synchronization operations, and block 3 follows the second synchronization operation, which is a release.

Release Consistency: Release consistency observes that weak ordering does not go far enough. It extends the weak ordering model by distinguishing among the types of synchronization operations and exploiting their associated semantics. In particular, it divides synchronization operations into acquires and releases. An *acquire* is a read operation (it can also be a read-modify-write) that is performed to gain access to a set of variables. Examples include the `Lock(TaskQ)` operation in part (a) of Figure 9-5, and the accesses to `flag` variables within the while conditions in part (b). A *release* is a write operation (or a read-modify-write) that grants permission to another processor to gain access to some variables. Examples include the "UnLock(TaskQ)" operation in part (a) of Figure 9-5, and the statements setting the `flag` variables to 1 in part (b).[1]

The separation into acquire and release operations can be used to further relax ordering constraints as shown in Figure 9-6. The purpose of an acquire is to delay memory accesses that follow the acquire statement until the acquire succeeds. It has nothing to do with accesses that precede it in program order (accesses in the block labeled "1"), so there is no reason to wait for those accesses to complete before the acquire can be issued. That is, the acquire itself can be reordered with respect to previous accesses. Similarly, the purpose of a release operation is to grant access to the new values of data modified before it in program order. It has nothing to do with accesses that follow it (accesses in the block labeled "3"), so these need not be delayed until the release has completed. However, we must wait for accesses in block "1" to complete before the release is visible to other processors (since we do not know exactly which variables are associated with the release or that the release "protects"[2]), and similarly we must wait for the acquire

1. Exercise: In Figure 9-5 are the statements setting Flag variables to 0 synchronization accesses?

2. It is possible that the release also grants access to variables outside of the variables controlled by the preceding acquire. The exact association of variables with synchronization accesses is very difficult to exploit at the hardware level. Software implementations of relaxed models, however, do exploit such optimizations, as we shall see in Section 9.4.

to complete before the operations in block "3" can be performed. Thus, the sufficient conditions for providing an RC interface are as follows: Before an operation labeled as a release is issued, the processor waits until all previous operations in program order have completed; and operations that follow an acquire operation in program order are not issued until that acquire operation completes. These are sufficient conditions, and we will see how we might be more aggressive when we discuss alternative approaches to a shared address space that rely on relaxed consistency models for performance.

Digital Alpha, Sparc V9 RMO, and IBM PowerPC memory models. The WO and RC models are specified in terms of using labeled synchronization operations to enforce orders, but do not take a position on the exact operations (instructions) that must be used. The memory models of some commercial microprocessors provide no ordering guarantees by default, but provide specific hardware instructions called memory barriers or *fences* that can be used to enforce orderings. To implement WO or RC with these microprocessors, operations that the WO or RC program labels as synchronizations (or acquires or releases) cause the compiler to insert the appropriate special instructions, or the programer can insert these instructions directly.

The Alpha architecture [Sit92], for example, supports two kinds of fence instructions: (i) the

Table 9-1 Characteristics of various system-centric relaxed models. A "yes" in the appropriate column indicates that those orders can be violated by that system-centric model.

Model	W-to-R reorder	W-to-W reorder	R-to-R/W reorder	Read Other's Write Early	Read Own Write Early	Operations for Ordering
SC					yes	
TSO	yes				yes	MEMBAR, RMW
PC	yes			yes	yes	MEMBAR, RMW
PSO	yes	yes			yes	STBAR, RMW
WO	yes	yes	yes		yes	SYNC
RC	yes	yes	yes	yes	yes	REL, ACQ, RMW
RMO	yes	yes	yes		yes	various MEMBARs
Alpha	yes	yes	yes		yes	MB, WMB
PowerPC	yes	yes	yes	yes	yes	SYNC

memory barrier (MB) and (ii) the write memory barrier (WMB). The MB fence is like a synchronization operation in WO: It waits for all previously issued memory accesses to complete before issuing any new accesses. The WMB fence imposes program order only between writes (it is like the STBAR in PSO). Thus, a read issued after a WMB can still bypass a write access issued before the WMB, but a write access issued after the WMB cannot. The Sparc V9 relaxed memory order (RMO) [WeG94] provides a fence instruction with four flavor bits associated with it. Each bit indicates a particular type of ordering to be enforced between previous and following load/store operations (the four possibilities are read-to-read, read-to-write, write-to-read, and write-to-write orderings). Any combinations of these bits can be set, offering a variety of ordering choices. Finally, the IBM PowerPC model [MSS+94,CSB93] provides only a single fence instruction, called SYNC, that is equivalent to Alpha's MB fence. It also differs from the Alpha and RMO models in that the writes are not atomic, as in the processor consistency model. The

model envisioned is WO, to be synthesized by putting SYNC instructions before and after every synchronization operation. You will see how different models can be synthesized with these primitives in Exercise 9.3.

The prominent specifications discussed above are summarized in Table 9-1.[1] They have different performance implications, and require different kinds of annotations to ensure orderings. It is worth noting again that although several of the models allow some reordering of operations, if program order is defined as seen by the programmer then only the models that allow both read and write operations to be reordered within sections of code (WO, RC, Alpha, RMO, and PowerPC) allow the flexibility needed by many important compiler optimizations. This may change if substantial improvements are made in compiler analysis to determine what reorderings are possible given a consistency model. The portability problem of these specifications should also be clear: for example, a program with enough memory barriers to work "correctly" (produce intuitive or sequentially consistent executions) on a TSO system will not necessarily work "correctly" when run on an RMO system: It will need more special operations. Let us therefore examine higher-level interfaces that are convenient for programmers and portable to the different systems, in each case safely exploiting the performance benefits and reorderings that the system affords.

9.2.2 The Programming Interface

The programming interfaces are inspired by the WO and RC models, in that they assume that program orders do not have to be maintained at all between synchronization operations. The idea is for the program to ensure that all synchronization operations are explicitly labeled or identified as such. The compiler or runtime library translates these synchronization operations into the appropriate order-preserving operations (memory barriers or fences) called for by the system specification. Then, the system (compiler plus hardware) guarantees sequentially consistent executions even though it may reorder operations between synchronization operations in any way it desires (without violating dependences to a location within a process). This allows the compiler sufficient flexibility between synchronization points for the reorderings it desires, and also allows the processor to perform as many reorderings as permitted by its memory model: If SC executions are guaranteed even with the weaker models that allow all reorderings, they surely will be guaranteed on systems that allow fewer reorderings. The consistency model presented at the programmer's interface should be at least as weak as that at the hardware interface, but need not be the same.

Programs that label all synchronization events are called *synchronized programs*. Formal models for specifying synchronized programs have been developed; namely the *data-race-free* models influenced by weak ordering [AdH90a] and the *properly-labeled* model [GAG+92] influenced by release consistency [GLL+90]. Interested readers can obtain more details from these references (the differences between them are small). The basic question for the programmer is how to decide which operations to label as synchronization. This is of course already done in the major-

1. The relaxation "read own write early" relates to both program order and write atomicity. The processor is allowed to read its own previous write before the write is serialized with respect to other writes to the same location. A common hardware optimization that relies on this relaxation is the processor reading the value of a variable from its own write buffer. This relaxation applies to almost all models without violating their semantics. It even applies to SC, as long as other program order and atomicity requirements are maintained.

ity of cases, when explicit, system-specified primitives such as locks and barriers are used. These are usually also easy to distinguish as acquire or release, for memory models such as RC that can take advantage of this distinction; for example, a lock is an acquire and an unlock is a release, and a barrier contains both since arrival at a barrier is a release (indicates completion of previous accesses) while leaving it is an acquire (obtaining permission for the new set of accesses). The real question is how to determine which memory operations on ordinary variables (such as our flag variables) should be labeled as synchronization. Often, programmers can identify these easily, since they know when they are using this event synchronization idiom. The following definitions describe a more general method when all else fails:

Conflicting operations: Two memory operations from different processes are said to *conflict* if they access the same memory location and at least one of them is a write.

Competing operations: These are a subset of the conflicting operations. Two conflicting memory operations (from different processes) are said to be competing if it is possible for them to appear next to each other in a sequentially consistent total order (execution); that is, to appear one immediately following the other in such an order, with no intervening memory operations on shared data between them.

Synchronized Program: A parallel program is *synchronized* if all competing memory operations have been labeled as synchronization (perhaps differentiated into acquire and release by labeling the read operations as acquires and the write operations as releases).

The fact that competing means competing under any possible SC interleaving is an important aspect of the programming interface. Even though the system uses a relaxed consistency model, the reasoning about where annotations are needed can itself be done while assuming an intuitive, SC execution model, shielding the programmer from reasoning directly in terms of reorderings. Of course, if the compiler could automatically determine what operations are conflicting or competing, then the programmer's task would be made a lot simpler. However, since the known analysis techniques are expensive and/or conservative [ShS88, KrY94], the job is almost always left to the programmer.

Consider the example in Figure 9-7. The accesses to the variable `flag` are competing operations

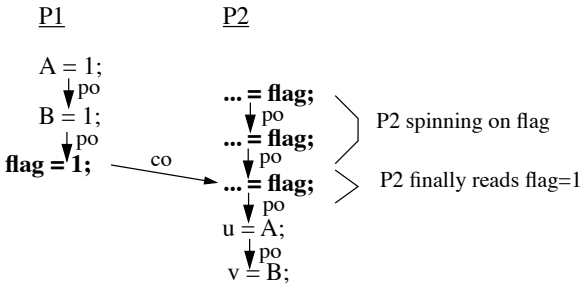

Figure 9-7 An example code sequence using spinning on a flag to synchronize producer-consumer behavior.

The arcs labeled "po" show program order, and that labeled "co" shows conflict order. Notice that between write to A by P1 and read to A by P2, there is a chain of accesses formed by po and co arcs. This will be true in *all* executions of the program. Such chains which have at least one po arc, will not be present for accesses to the variable flag (there is only a co arc).

by the above definition. What this really means is that on a multiprocessor they may execute simultaneously on their respective processors, and we have no guarantee about which executes first. Thus, they are also said to constitute *data races*. In contrast, the accesses to variable `A` (and

to B) by P1 and P2 are conflicting operations, but they are necessarily separated in any SC interleaving by an intervening write to the variable flag by P1 and a corresponding read of flag by P2. Thus, they are not competing accesses and do not have to be labeled as synchronization.

To be a little more formal, given a particular SC execution order the *conflict order* (co) is the order in which the conflicting operations occur (note that in a given total or execution order one must occur before the other). In addition, we have the program order (po) for each process. Figure 9-7 also shows the program orders and conflict order for a sample execution of our code fragment. Two accesses are non-competing if under all possible SC executions (interleavings) there always exists a chain of other references between them, such that at least one link in the chain is formed by a program-order rather than conflict order arc. The complete formalism can be found in the literature [Gha95].

Of course, the definition of synchronized programs allows a programmer to conservatively label more operations than necessary as synchronization, without compromising correctness. In the extreme, labeling all memory operations as synchronization always yields a synchronized program. This extreme example will of course deny us the performance benefits that the system might otherwise provide by reordering non-synchronization operations, and will on most systems yield much worse performance than straightforward SC implementations due to the overhead of the order-preserving instructions that will be inserted. The goal is to only label competing operations as synchronization.

In specific circumstances, we may decide not to label some competing accesses as synchronization, since we want to allow data races in the program. Now we are no longer guaranteed SC semantics, but we may know through application knowledge that the competing operations are not being used as synchronization and we do not need such strong ordering guarantees in certain sections of code. An example is the use of the asynchronous rather than red-black equation solver in Chapter 2. There is no synchronization between barriers (sweeps) so within a sweep the read and write accesses to the border elements of a partition are competing accesses. The program will not satisfy SC semantics on a system that allows access reorderings, but this is okay since the solver repeats the sweeps until convergence: Even if the processes sometimes read old values in an iteration and sometimes new (unpredictably), they will ultimately read updated values and make progress toward convergence. If we had labeled the competing accesses, we would have compromised reordering and performance.

The last issue related to the programming interface is how the above labels are to be specified by the programmer. In many cases, this is quite stylized and already present in the programming language. For example, some parallel programming languages (e.g., High-Performance Fortran [HPF93] allow parallelism to be expressed in only stylized ways from which it is trivial to extract the relevant information. For example, in FORALL loops (loops in which all iterations are independent) only the implicit barrier at the end of the loop needs to be labeled as synchronization: The FORALL specifies that there are no data races within the loop body. In more general programming models, if programmers use a library of synchronization primitives such as LOCK, UNLOCK and BARRIER, then the code that implements these primitives can be labeled by the designer of the library; the programmer needn't do anything special. Finally, if the application programmer wants to add further labels at memory operations—for example at flag variable accesses or to preserve some other orders as in the examples of Figure 9-4 on page 621—we need programming language or library support. A programming language could provide an attribute for variable declarations that indicates that all references to this variable are synchronization accesses, or there could be annotations at the statement level, indicating that a particular

access is to be labeled as a synchronization operation. This tells the compiler to constrain its reordering across those points, and the compiler in turn translates these references to the appropriate order-preserving mechanisms for the processor.

9.2.3 Translation Mechanisms

For most microprocessors, translating labels to order preserving mechanisms amounts to inserting a suitable memory barrier instruction before and/or after each operation labeled as a synchronization (or acquire or release). It would save instructions if one could have flavor bits associated with individual loads/stores themselves, indicating what orderings to enforce and avoiding extra instructions, but since the operations are usually infrequent this is not the direction that most microprocessors have taken so far.

9.2.4 Consistency Models in Real Systems

With the large growth in sales of multiprocessors, modern microprocessors are designed so that they can be seamlessly integrated into these machines. As a result, microprocessor vendors expend substantial effort defining and precisely specifying the memory model presented at the hardware-software interface. While sequential consistency remains the best model to reason about correctness and semantics of programs, for performance reasons many vendors allow orders to be relaxed. Some vendors like Silicon Graphics (in the MIPS R10000) continue to support SC by allowing out-of-order issue and execution of operations, but not out of order completion or visibility. This allows overlapping of memory operations by the dynamically scheduled processor, but forces them to complete in order. The Intel Pentium family supports a processor consistency model, so reads can complete before previous writes in program order (see illustration), and many microprocessors from SUN Microsystems support TSO which allows the same reorderings. Many other vendors have moved to models that allow all orders to be relaxed (e.g., Digital Alpha and IBM PowerPC) and provide memory barriers to enforce orderings where necessary.

At the hardware interface, multiprocessors usually follow the consistency model exported by the microprocessors they use, since this is the easiest thing to do. For example, we saw that the NUMA-Q system exports the processor consistency model of its Pentium Pro processors at this interface. In particular, on a write the ownership and/or data are obtained before the invalidations begin. The processor is allowed to complete its write and go on as soon as the ownership/data is received, and the SCLIC takes care of the invalidation/acknowledgment sequence. It is also possible for the communication assist to alter the model, within limits, but this comes at the cost of design complexity. We have seen an example in the Origin2000, where preserving SC requires that the assist (Hub) only reply to the processor on a write once the exclusive reply and all invalidation acknowledgments have been received (called *delayed* exclusive replies). The dynamically scheduled processor can then retire its write from the instruction lookahead buffer and allow subsequent operations to retire and complete as well. If the Hub replies as soon as the exclusive reply is received and before invalidation acknowledgments are received (called *eager* exclusive replies), then the write will retire and subsequent operations complete before the write is actually completed, and the consistency model is more relaxed. Essentially, the Hub fools the processor about the completion of the write.

Having the assist cheat the processor can enhance performance but increases complexity, as in the case of eager exclusive replies. The handling of invalidation acknowledgments must now be

done asynchronously by the Hub through tracking buffers in its processor interface, in accordance with the desired relaxed consistency model. There are also questions about whether subsequent accesses to a block from other processors, forwarded to this processor from the home, should be serviced while invalidations for a write on that block are still outstanding (see Exercise 9.2), and what happens if the processor has to write that block back to memory due to replacement while invalidations are still outstanding in the Hub. In the latter case, either the writeback has to be buffered by the Hub and delayed till all invalidations are acknowledged, or the protocol must be extended so a later access to the written back block is not satisfied by the home until the acknowledgments are received by the requestor. The extra complexity and the perceived lack of substantial performance improvement led the Origin designers to persist with a sequential consistency model, and reply to the processor only when all acknowledgments are received.

On the compiler side, the picture for memory consistency models is not so well defined, complicating matters for programmers. It does not do a programmer much good for the processor to support sequential consistency or processor consistency if the compiler reorders accesses as it pleases before they even get to the processor (as uniprocessor compilers do). Microprocessor memory models are defined at the hardware interface; they tend to be concerned with program order as presented to the processor, and assume that a separate arrangement will be made with the compiler. As we have discussed, exporting intermediate models such as TSO, processor consistency and PSO up to the programmer's interface does not allow the compiler enough flexibility for reordering. We might assume that the compiler most often will not reorder the operations that would violate the consistency model, e.g. since most compiler reorderings of memory operations tend to focus on loops, but this is a very dangerous assumption, and sometimes the orders we rely upon indeed occur in loops. To really use these models at the programmer's interface, uniprocessor compilers would have to be modified to follow these reordering specifications, compromising performance significantly. An alternative approach, mentioned earlier, is to use compiler analysis to determine when operations can be reordered without compromising the consistency model exposed to the user [ShS88, KrY94]. However, the compiler analysis must be able to detect conflicting operations from different processes, which is expensive and currently rather conservative.

More relaxed models like Alpha, RMO, PowerPC, WO, RC and synchronized programs can be used at the programmer's interface because they allow the compiler the flexibility it needs. In these cases, the mechanisms used to communicate ordering constraints must be heeded not only by the processor but also by the compiler. Compilers for multiprocessors are beginning to do so (see also Section 9.6). Underneath a relaxed model at the programmer's interface, the processor interface can use the same or a stronger ordering model, as we saw in the context of synchronized programs. However, there is now significant motivation to use relaxed models even at the processor interface, to realize the performance potential. As we move now to discussing alternative approaches to supporting a shared address space with coherent replication of data, we shall see that relaxed consistency models can be critical to performance when we want to support coherence at larger granularities than cache blocks. We will also see that the consistency model can be relaxed even beyond release consistency (can you think how?).

9.3 Overcoming Capacity Limitations

The systems that we have discussed so far provide automatic replication and coherence in hardware only in the processor caches. A processor cache replicates remotely allocated data directly