8.5 Synchronization

Synchronization mechanisms are typically built with user-level software routines that rely on hardware-supplied synchronization instructions. For smaller machines or low-contention situations, the key hardware capability is an uninterruptible instruction or instruction sequence capable of atomically retrieving and changing a value. Software synchronization mechanisms are then constructed using this capability. For example, we will see how very efficient spin locks can be built using a simple hardware synchronization instruction and the coherence mechanism. In larger-scale machines or high-contention situations, synchronization can become a performance bottleneck, because contention introduces additional delays and because latency is potentially greater in such a machine. We will see how contention can arise in implementing some common user-level synchronization operations and examine more powerful hardware-supported synchronization primitives that can reduce contention as well as latency.

We begin by examining the basic hardware primitives, then construct several well-known synchronization routines with the primitives, and then turn to performance problems in larger machines and solutions for those problems.

Basic Hardware Primitives

The key ability we require to implement synchronization in a multiprocessor is a set of hardware primitives with the ability to atomically read and modify a memory location. Without such a capability, the cost of building basic synchronization primitives will be too high and will increase as the processor count increases. There are a number of alternative formulations of the basic hardware primitives, all of which provide the ability to atomically read and modify a location, together with some way to tell if the read and write were performed atomically. These hardware primitives are the basic building blocks that are used to build a wide variety of user-level synchronization operations, including things such as locks and barriers. In general, architects do not expect users to employ the basic hardware primitives, but instead expect that the primitives will be used by system programmers to build a synchronization library, a process that is often complex and tricky. Let's start with one such hardware primitive and show how it can be used to build some basic synchronization operations.

One typical operation for building synchronization operations is the *atomic exchange*, which interchanges a value in a register for a value in memory. To see how to use this to build a basic synchronization operation, assume that we want to build a simple lock where the value 0 is used to indicate that the lock is free and a 1 is used to indicate that the lock is unavailable. A processor tries to set the lock by doing an exchange of 1, which is in a register, with the memory address corresponding to the lock.The value returned from the exchange instruction is 1 if some other processor had already claimed access and 0 otherwise. In the latter case, the value is also changed to be 1, preventing any competing exchange from also retrieving a 0.

For example, consider two processors that each try to do the exchange simultaneously: This race is broken since exactly one of the processors will perform the exchange first, returning 0, and the second processor will return 1 when it does the exchange. The key to using the exchange (or swap) primitive to implement synchronization is that the operation is atomic: the exchange is indivisible and two simultaneous exchanges will be ordered by the write serialization mechanisms. It is impossible for two processors trying to set the synchronization variable in this manner to both think they have simultaneously set the variable.

There are a number of other atomic primitives that can be used to implement synchronization. They all have the key property that they read and update a memory value in such a manner that we can tell whether or not the two operations executed atomically. One operation present in many older machines is *test-and-set,* which tests a value and sets it if the value passes the test. For example, we could define an operation that tested for 0 and set the value to 1, which can be used in a fashion similar to how we used atomic exchange. Another atomic synchronization primitive is *fetch-and-increment:* it returns the value of a memory location and atomically increments it. By using the value 0 to indicate that the synchronization variable is unclaimed, we can use fetch-and-increment, just as we used exchange. There are other uses of operations like fetch-and-increment, which we will see shortly.

A slightly different approach to providing this atomic read-and-update operation has been used in some recent machines. Implementing a single atomic memory operation introduces some challenges, since it requires both a memory read and a write in a single, uninterruptible instruction. This complicates the implementation of coherence, since the hardware cannot allow any other operations between the read and the write, and yet must not deadlock.

An alternative is to have a pair of instructions where the second instruction returns a value from which it can be deduced whether the pair of instructions was executed as if the instructions were atomic. The pair of instructions appears atomic if it appears as if all other operations executed by any processor appear before or after the pair. Thus when an instruction pair appears atomic, no other processor can change the value between the instruction pair.

The pair of instructions includes a special load called a *load linked* or *load locked* and a special store called a *store conditional.* These instructions are used in sequence: If the contents of the memory location specified by the load linked are changed before the store conditional to the same address occurs, then the store conditional fails. If the processor does a context switch between the two instructions, then the store conditional also fails. The store conditional is defined to return a value indicating whether or not the store was successful. Since the load linked returns the initial value and the store conditional returns 1 if it succeeds

and 0 otherwise, the following sequence implements an atomic exchange on the memory location specified by the contents of R1:

```
try:  mov   R3,R4        ;mov exchange value
      ll    R2,0(R1)     ;load linked
      sc    R3,0(R1)     ;store conditional
      beqz  R3,try       ;branch store fails
      mov   R4,R2        ;put load value in R4
```

At the end of this sequence the contents of R4 and the memory location specified by R1 have been atomically exchanged. Any time a processor intervenes and modifies the value in memory between the `ll` and `sc` instructions, the `sc` returns 0 in R3, causing the code sequence to try again.

An advantage of the load linked/store conditional mechanism is that it can be used to build other synchronization primitives. For example, here is an atomic fetch-and-increment:

```
try:  ll    R2,0(R1)     ;load linked
      addi  R2,R2,#1     ;increment
      sc    R2,0(R1)     ;store conditional
      beqz  R2,try       ;branch store fails
```

These instructions are typically implemented by keeping track of the address specified in the `ll` instruction in a register, often called the *link register*. If an interrupt occurs, or if the cache block matching the address in the link register is invalidated (for example, by another `sc`), the link register is cleared. The `sc` instruction simply checks that its address matches that in the link register; if so, the `sc` succeeds; otherwise, it fails. Since the store conditional will fail after either another attempted store to the load linked address or any exception, care must be taken in choosing what instructions are inserted between the two instructions. In particular, only register-register instructions can safely be permitted; otherwise, it is possible to create deadlock situations where the processor can never complete the `sc`. In addition, the number of instructions between the load linked and the store conditional should be small to minimize the probability that either an unrelated event or a competing processor causes the store conditional to fail frequently.

Implementing Locks Using Coherence

Once we have an atomic operation, we can use the coherence mechanisms of a multiprocessor to implement *spin locks:* locks that a processor continuously tries to acquire, spinning around a loop. Spin locks are used when we expect the lock to be held for a very short amount of time and when we want the process of locking to be low latency when the lock is available. Because spin locks tie up the

processor, waiting in a loop for the lock to become free, they are inappropriate in some circumstances.

The simplest implementation, which we would use if there were no cache coherence, would keep the lock variables in memory. A processor could continually try to acquire the lock using an atomic operation, say exchange, and test whether the exchange returned the lock as free. To release the lock, the processor simply stores the value 0 to the lock. Here is the code sequence to lock a spin lock whose address is in R1 using an atomic exchange:

```
      li    R2,#1
lockit: exch  R2,0(R1)   ;atomic exchange
      bnez  R2,lockit    ;already locked?
```

If our machine supports cache coherence, we can cache the locks using the coherence mechanism to maintain the lock value coherently. This has two advantages. First, it allows an implementation where the process of "spinning" (trying to test and acquire the lock in a tight loop) could be done on a local cached copy rather than requiring a global memory access on each attempt to acquire the lock. The second advantage comes from the observation that there is often locality in lock accesses: that is, the processor that used the lock last will use it again in the near future. In such cases, the lock value may reside in the cache of that processor, greatly reducing the time to acquire the lock.

To obtain the first advantage—being able to spin on a local cached copy rather than generating a memory request for each attempt to acquire the lock—requires a change in our simple spin procedure. Each attempt to exchange in the loop directly above requires a write operation. If multiple processors are attempting to get the lock, each will generate the write. Most of these writes will lead to write misses, since each processor is trying to obtain the lock variable in an exclusive state.

Thus we should modify our spin-lock procedure so that it spins by doing reads on a local copy of the lock until it successfully sees that the lock is available. Then it attempts to acquire the lock by doing a swap operation. A processor first reads the lock variable to test its state. A processor keeps reading and testing until the value of the read indicates that the lock is unlocked. The processor then races against all other processes that were similarly "spin waiting" to see who can lock the variable first. All processes use a swap instruction that reads the old value and stores a 1 into the lock variable. The single winner will see the 0, and the losers will see a 1 that was placed there by the winner. (The losers will continue to set the variable to the locked value, but that doesn't matter.) The winning processor executes the code after the lock and, when finished, stores a 0 into the lock variable to release the lock, which starts the race all over again. Here is the code to perform this spin lock (remember that 0 is unlocked and 1 is locked):

```
lockit: lw    R2,0(R1)    ;load of lock
        bnez  R2,lockit   ;not available-spin
        li    R2,#1       ;load locked value
        exch  R2,0(R1)    ;swap
        bnez  R2,lockit   ;branch if lock wasn't 0
```

Let's examine how this "spin-lock" scheme uses the cache-coherence mechanisms. Figure 8.32 shows the processor and bus or directory operations for multiple processes trying to lock a variable using an atomic swap. Once the processor with the lock stores a 0 into the lock, all other caches are invalidated and must fetch the new value to update their copy of the lock. One such cache gets the copy of the unlocked value (0) first and performs the swap. When the cache miss of other processors is satisfied, they find that the variable is already locked, so they must return to testing and spinning.

Step	Processor P0	Processor P1	Processor P2	Coherence state of lock	Bus/directory activity
1	Has lock	Spins, testing if lock = 0	Spins, testing if lock = 0	Shared	None
2	Set lock to 0	(Invalidate received)	(Invalidate received)	Exclusive	Write invalidate of lock variable from P0
3		Cache miss	Cache miss	Shared	Bus/directory services P2 cache miss; write back from P0
4		(Waits while bus/ directory busy)	Lock = 0	Shared	Cache miss for P2 satisfied
5		Lock = 0	Executes swap, gets cache miss	Shared	Cache miss for P1 satisfied
6		Executes swap, gets cache miss	Completes swap: returns 0 and sets Lock =1	Exclusive	Bus/directory services P2 cache miss; generates invalidate
7		Swap completes and returns 1	Enter critical section	Shared	Bus/directory services P2 cache miss; generates write back
8		Spins, testing if lock = 0			None

FIGURE 8.32 Cache-coherence steps and bus traffic for three processors, P0, P1, and P2. This figure assumes write-invalidate coherence. P0 starts with the lock (step 1). P0 exits and unlocks the lock (step 2). P1 and P2 race to see which reads the unlocked value during the swap (steps 3–5). P2 wins and enters the critical section (steps 6 and 7), while P1's attempt fails so it starts spin waiting (steps 7 and 8). In a real system, these events will take many more than eight clock ticks, since acquiring the bus and replying to misses takes much longer.

This example shows another advantage of the load-linked/store-conditional primitives: the read and write operation are explicitly separated. The load linked need not cause any bus traffic. This allows the following simple code sequence, which has the same characteristics as the optimized version using exchange (R1 has the address of the lock):

```
lockit: ll    R2,0(R1)    ;load linked
        bnez  R2,lockit   ;not available-spin
        li    R2,#1       ;locked value
        sc    R2,0(R1)    ;store
        beqz  R2,lockit   ;branch if store fails
```

The first branch forms the spinning loop; the second branch resolves races when two processors see the lock available simultaneously.

Although our spin lock scheme is simple and compelling, it has difficulty scaling up to handle many processors because of the communication traffic generated when the lock is released. The next section discusses these problems in more detail, as well as techniques to overcome these problems in larger machines.

Synchronization Performance Challenges

To understand why the simple spin-lock scheme of the previous section does not scale well, imagine a large machine with all processors contending for the same lock. The directory or bus acts as a point of serialization for all the processors, leading to lots of contention, as well as traffic. The following Example shows how bad things can be.

EXAMPLE Suppose there are 20 processors on a bus that each try to lock a variable simultaneously. Assume that each bus transaction (read miss or write miss) is 50 clock cycles long. You can ignore the time of the actual read or write of a lock held in the cache, as well as the time the lock is held (they won't matter much!). Determine the number of bus transactions required for all 20 processors to acquire the lock, assuming they are all spinning when the lock is released at time 0. About how long will it take to process the 20 requests? Assume that the bus is totally fair so that every pending request is serviced before a new request and that the processors are equally fast.

ANSWER Figure 8.33 shows the sequence of events from the time of the release to the time to the next release. Of course, the number of processors contending for the lock drops by one each time the lock is acquired, which reduces the average cost to 1525 cycles. Thus for 20 lock-unlock pairs it will

take over 30,000 cycles for the processors to pass through the lock. Furthermore, the average processor will spend half this time idle, simply trying to get the lock. The number of bus transactions involved is over 400!

Event	Duration
Read miss by all waiting processors to fetch lock (20×50)	1000
Write miss by releasing processor and invalidates	50
Read miss by all waiting processors (20×50)	1000
Write miss by all waiting processors, one successful lock (50), and invalidation of all lock copies (19×50)	1000
Total time for one processor to acquire and release lock	3050 clocks

FIGURE 8.33 The time to acquire and release a single lock when 20 processors contend for the lock, assuming each bus transaction takes 50 clock cycles. Because of fair bus arbitration, the releasing processor must wait for *all* other 19 processors to try to get the lock in vain!

■

The difficulty in this Example arises from contention for the lock and serialization of lock access, as well as the latency of the bus access. The fairness property of the bus actually makes things worse, since it delays the processor that claims the lock from releasing it; unfortunately, for any bus arbitration scheme some worst-case scenario does exist. The root of the problem is the contention and the fact that the lock access is serialized. The key advantages of spin locks, namely that they have low overhead in terms of bus or network cycles and offer good performance when locks are reused by the same processor, are both lost in this example. We will consider alternative implementations in the next section, but before we do that, let's consider the use of spin locks to implement another common high-level synchronization primitive.

Barrier Synchronization

One additional common synchronization operation in programs with parallel loops is a *barrier*. A barrier forces all processes to wait until all the processes reach the barrier and then releases all of the processes. A typical implementation of a barrier can be done with two spin locks: one used to protect a counter that tallies the processes arriving at the barrier and one used to hold the processes until the last process arrives at the barrier. To implement a barrier we usually use the ability to spin on a variable until it satisfies a test; we use the notation `spin(condition)` to indicate this. Figure 8.34 is a typical implementation, assuming that lock and unlock provide basic spin locks and `total` is the number of processes that must reach the barrier.


```
lock (counterlock);/* ensure update atomic */
if (count==0) release=0;/*first=>reset release */
count = count +1;/* count arrivals */
unlock(counterlock);/* release lock */
if (count==total) { /* all arrived */
       count=0;/* reset counter */
       release=1;/* release processes */
}
else { /* more to come */

       spin (release=1);/* wait for arrivals */
}
```

FIGURE 8.34 Code for a simple barrier. The lock `counterlock` protects the counter so that it can be atomically incremented. The variable `count` keeps the tally of how many processes have reached the barrier. The variable `release` is used to hold the processes until the last one reaches the barrier.The operation `spin (release=1)` causes a process to wait until all processes reach the barrier.

In practice, another complication makes barrier implementation slightly more complex. Frequently a barrier is used within a loop, so that processes released from the barrier would do some work and then reach the barrier again. Assume that one of the processes never actually leaves the barrier (it stays at the spin operation), which could happen if the OS scheduled another process, for example. Now it is possible that one process races ahead and gets to the barrier again before the last process has left. The fast process traps that last slow process in the barrier by resetting the flag `release`. Now all the processes will wait infinitely at the next instance of this barrier, because one process is trapped at the last instance, and the number of processes can never reach the value of total. The important observation is that the programmer did nothing wrong. Instead, the implementer of the barrier made some assumptions about forward progress that cannot be assumed. One obvious solution to this is to count the processes as they exit the barrier (just as we did on entry) and not to allow any process to reenter and reinitialize the barrier until all processes have left the prior instance of this barrier. This would significantly increase the latency of the barrier and the contention, which as we will see shortly are already large. An alternative solution is a *sense-reversing barrier,* which makes use of a private per-process variable, `local_sense`, which is initialized to 1 for each process. Figure 8.35 shows the code for the sense-reversing barrier. This version of a barrier is safely usable; however, as the next Example shows, its performance can still be quite poor.

```
local_sense = ! local_sense; /*toggle local_sense*/
lock (counterlock);/* ensure update atomic */
count++;/* count arrivals */
unlock (counterlock);/* unlock */
if (count==total) { /* all arrived */
      count=0;/* reset counter */
      release=local_sense;/* release processes */
}
else { /* more to come */
      spin (release=local_sense);/*wait for signal*/
}
```

FIGURE 8.35 Code for a sense-reversing barrier. The key to making the barrier reusable is the use of an alternating pattern of values for the flag release, which controls the exit from the barrier. If a process races ahead to the next instance of this barrier while some other processes are still in the barrier, the fast process cannot trap the other processes, since it does not reset the value of `release` as it did in Figure 8.34.

EXAMPLE Suppose there are 20 processors on a bus that each try to execute a barrier simultaneously. Assume that each bus transaction is 50 clock cycles, as before. You can ignore the time of the actual read or write of a lock held in the cache as the time to execute other nonsynchronization operations in the barrier implementation. Determine the number of bus transactions required for all 20 processors to reach the barrier, be released from the barrier, and exit the barrier. Assume that the bus is totally fair, so that every pending request is serviced before a new request and that the processors are equally fast. Don't worry about counting the processors out of the barrier. How long will the entire process take?

ANSWER The following table shows the sequence of events for one processor to traverse the barrier, assuming that the first process to grab the bus does not have the lock.

Event	Duration in clocks for one processor	Duration in clocks for 20 processors
Time for each processor to grab lock, increment, release lock	1525	30,500
Time to execute release	50	50
Time for each processor to get the release flag	50	1000
Total	1625	31,550

Our barrier operation takes a little longer than the 20-processor lock-unlock sequence we considered earlier. The total number of bus transactions is about 440. ■

As we can see from these Examples, synchronization performance can be a real bottleneck when there is substantial contention among multiple processes. When there is little contention and synchronization operations are infrequent, we are primarily concerned about the latency of a synchronization primitive—that is, how long it takes an individual process to complete a synchronization operation. Our basic spin-lock operation can do this in two bus cycles: one to initially read the lock and one to write it. We could improve this to a single bus cycle by a variety of methods. For example, we could simply spin on the swap operation. If the lock were almost always free, this could be better, but if the lock were not free, it would lead to lots of bus traffic, since each attempt to lock the variable would lead to a bus cycle. In practice, the latency of our spin lock is not quite as bad as we have seen in this Example, since the write miss for a data item present in the cache is treated as an upgrade and will be cheaper than a true read miss.

The more serious problem in these Examples is the serialization of each process' attempt to complete the synchronization. This serialization is a problem when there is contention, because it greatly increases the time to complete the synchronization operation. For example, if the time to complete all 20 lock and unlock operations depended only on the latency in the uncontended case, then it would take 2000 rather than 40,000 cycles to complete the synchronization operations. The use of a bus interconnect exacerbates this problem, but serialization could be just as serious in a directory-based machine, where the latency would be large. The next section presents some solutions that are useful when either the contention is high or the processor count is large.

Synchronization Mechanisms for Larger-Scale Machines

What we would like are synchronization mechanisms that have low latency in uncontended cases and that minimize serialization in the case where contention is significant. We begin by showing how software implementations can improve the performance of locks and barriers when contention is high; we then explore two basic hardware primitives that reduce serialization while keeping latency low.

Software Implementations

The major difficulty with our spin-lock implementation is the delay due to contention when many processes are spinning on the lock. One solution is to artificially delay processes when they fail to acquire the lock. This is done by delaying attempts to reacquire the lock whenever the store-conditional operation fails. The best performance is obtained by increasing the delay exponentially whenever the

attempt to acquire the lock fails. Figure 8.36 shows how a spin lock with *exponential back-off* is implemented. Exponential back-off is a common technique for reducing contention in shared resources, including access to shared networks and buses (see section 7.7). This implementation still attempts to preserve low latency when contention is small by not delaying the initial spin loop. The result is that if many processes are waiting, the back-off does not affect the processes on their first attempt to acquire the lock. We could also delay that process, but the result would be poorer performance when the lock was in use by only two processes and the first one happened to find it locked.

```
        li    R3,1        ;R3 = initial delay
lockit: ll    R2,0(R1)    ;load linked
        bnez  R2,lockit   ;not available-spin
        addi  R2,R2,1     ;get locked value
        sc    R2,0(R1)    ;store conditional
        bnez  R2,gotit    ;branch if store succeeds
        sll   R3,R3,1     ;increase delay by 2
        pause R3          ;delays by value in R3
        j     lockit
goit:   use data protected by lock
```

FIGURE 8.36 A spin lock with exponential back-off. When the store conditional fails, the process delays itself by the value in R3. The delay can be implemented by decrementing R3 until it reaches 0. The exact timing of the delay is machine dependent, although it should start with a value that is approximately the time to perform the critical section and release the lock. The statement `pause R3` should cause a delay of R3 of these time units. The value in R3 is increased by a factor of 2 every time the store conditional fails, which causes the process to wait twice as long before trying to acquire the lock again.

Another technique for implementing locks is to use queuing locks. We show how this works in the next section using a hardware implementation, but software implementations using arrays can achieve most of the same benefits (see Exercise 8.24). Before we look at hardware primitives, let's look at a better mechanism for barriers.

Our barrier implementation suffers from contention both during the *gather* stage, when we must atomically update the count, and at the *release* stage, when all the processes must read the release flag. The former is more serious because it requires exclusive access to the synchronization variable and thus creates much more serialization; in comparison, the latter generates only read contention. We can reduce the contention by using a *combining tree*, a structure where multiple requests are locally combined in tree fashion. The same combining tree can be used to implement the release process, reducing the contention there; we leave the last step for the Exercises.

Our combining tree barrier uses a predetermined n-ary tree structure. We use the variable k to stand for the fan-in; in practice $k = 4$ seems to work well. When the k-th process arrives at a node in the tree, we signal the next level in the tree. When a process arrives at the root, we release all waiting processes. As in our earlier example, we use a sense-reversing technique. The following tree-based barrier uses a tree to combine the processes and a single signal to release the barrier.

```
struct node{ /* a node in the combining tree */
   int counterlock; */ lock for this node */
   int count; */ counter for this node */
   int: parent; */ parent in the tree = 0..P-1 except for root
                  = -1*/
};
struct node tree [0..P-1]; */ the tree of nodes */
int local_sense; */ private per processor */
int release; */ global release flag */

*/ function to implement barrier */
barrier (int mynode) {
   lock (tree[mynode].counterlock); */ protect count */
   count++; */ increment count */
   unlock (tree[mynode].counterlock); */ unlock */
   if (tree[mynode].count==k) { */ all arrived at mynode */
      if (tree[mynode].parent >=0 {
         barrier(tree[mynode].parent);
      } else{
         release = local_sense;
      }
      tree[mynode].count = 0; */ reset for the next time */
   } else{
      spin (release=local_sense); /* wait */
   };
};
*/ code executed by a processor to join barrier */
local sense = | local_sense;
barrier (mynode);
```

The tree is assumed to be prebuilt statically using the nodes in the array `tree`. Each node in the tree combines k processes and provides a separate counter and lock, so that at most k processes contend at each node. When the kth process reaches a node in the tree it goes up to the parent, decrementing the count at the parent. When the count in the parent node reaches k, the release flag is set. The count in each node is reset by the last process to arrive. Sense-reversing is used to avoid races as in the simple barrier. Exercises 8.22 and 8.23 ask you to analyze

the time for the combining barrier versus the noncombining version. Some MPPs (e.g., the T3D and CM-5) have also included hardware support for barriers, but whether such facilities will be included in future machines is unclear.

Hardware Primitives

In this section we look at two hardware synchronization primitives. The first primitive deals with locks, while the second is useful for barriers and a number of other user-level operations that require counting or supplying distinct indices. In both cases we can create a hardware primitive where latency is essentially identical to our earlier version, but with much less serialization, leading to better scaling when there is contention.

The major problem with our original lock implementation is that it introduces a large amount of unneeded contention. For example, when the lock is released all processors generate both a read and a write miss, although at most one processor can successfully get the lock in the unlocked state. This happens on each of the 20 lock/unlock sequences. We can improve this situation by explicitly handing the lock from one waiting processor to the next. Rather than simply allowing all processors to compete every time the lock is released, we keep a list of the waiting processors and hand the lock to one explicitly, when its turn comes. This sort of mechanism has been called a *queuing lock*. Queuing locks can be implemented either in hardware, which we describe here, or in software using an array to keep track of the waiting processes. The basic concepts are the same in either case. Our hardware implementation assumes a directory-based machine where the individual processor caches are addressable. In a bus-based machine, a software implementation would be more appropriate and would have each processor using a different address for the lock, permitting the explicit transfer of the lock from one process to another.

How does a queuing lock work? On the first miss to the lock variable, the miss is sent to a synchronization controller, which may be integrated with the memory controller (in a bus-based system) or with the directory controller. If the lock is free, it is simply returned to the processor. If the lock is unavailable, the controller creates a record of the node's request (such as a bit in a vector) and sends the processor back a locked value for the variable, which the processor then spins on. When the lock is freed, the controller selects a processor to go ahead from the list of waiting processors. It can then either update the lock variable in the selected processor's cache or invalidate the copy, causing the processor to miss and fetch an available copy of the lock.

EXAMPLE How many bus transaction and how long does it take to have 20 processors lock and unlock the variable using a queuing lock that updates the lock on a miss? Make the other assumptions about the system the same as before.

ANSWER Each processor misses once on the lock initially and once to free the lock, so it takes only 40 bus cycles. The first 20 initial misses take 1000 cycles, followed by a 50-cycle delay for each of the 20 releases. This is a total of 2050 cycles—significantly better than the case with conventional coherence-based spin locks. ■

There are a couple of key insights in implementing such a queuing lock capability. First, we need to be able to distinguish the initial access to the lock, so we can perform the queuing operation, and also the lock release, so we can provide the lock to another processor. The queue of waiting processes can be implemented by a variety of mechanisms. In a directory-based machine, this queue is akin to the sharing set, and similar hardware can be used to implement the directory and queuing lock operations. One complication is that the hardware must be prepared to reclaim such locks, since the process that requested the lock may have been context-switched and may not even be scheduled again on the same processor.

Queuing locks can be used to improve the performance of our barrier operation (see Exercise 8.15). Alternatively, we can introduce a primitive that reduces the amount of time needed to increment the barrier count, thus reducing the serialization at this bottleneck, which should yield comparable performance to using queuing locks. One primitive that has been introduced for this and for building other synchronization operations is *fetch-and-increment,* which atomically fetches a variable and increments its value. The returned value can be either the incremented value or the fetched value. Using fetch-and-increment we can dramatically improve our barrier implementation, compared to the simple code-sensing barrier.

EXAMPLE Write the code for the barrier using fetch-and-increment. Making the same assumptions as in our earlier example and also assuming that a fetch-and-increment operation takes 50 clock cycles, determine the time for 20 processors to traverse the barrier. How many bus cycles are required?

ANSWER Figure 8.37 shows the code for the barrier. This implementation requires 20 fetch-and-increment operations and 20 cache misses for the release operation. This is a total time of 2000 cycles and 40 bus/interconnect operations versus an earlier implementation that took over 15 times longer and 10 times more bus operations to complete the barrier. Of course, fetch-and-increment can also be used in implementing the combining tree barrier, reducing the serialization at each node in the tree. ■

```
local_sense = ! local_sense; /*toggle local_sense*/
fetch_and_increment(count);/* atomic update*/
if (count==total) { /* all arrived */
      count=0;/* reset counter */
      release=local_sense;/* release processes */
}
else { /* more to come */
      spin (release=local_sense);/*wait for signal*/
}
```

FIGURE 8.37 Code for a sense-reversing barrier using fetch-and-increment to do the counting.

As we have seen, synchronization problems can become quite acute in larger-scale machines. When the challenges posed by synchronization are combined with the challenges posed by long memory latency and potential load imbalance in computations, we can see why getting efficient usage of large-scale parallel machines is very challenging. In section 8.10 we will examine the costs of synchronization on an existing bus-based multiprocessor for some real applications.

8.6 Models of Memory Consistency

Cache coherence ensures that multiple processors see a consistent view of memory. It does not answer the question of *how* consistent the view of memory must be. By this we mean, When must a processor see a value that has been updated by another processor?

Since processors communicate through shared variables (both those for data values and those used for synchronization), the question boils down to this: In what order must a processor observe the data writes of another processor?

Since the only way to "observe the writes of another processor" is through reads, the question becomes, What properties must be enforced among reads and writes to different locations by different processors?

Although the question, how consistent?, seems simple, it is remarkably complicated, as we can see in the following example. Here are two code segments from processes P1 and P2, shown side by side:

```
P1:   A = 0;              P2:   B = 0;
      .....                     .....
      A = 1;                    B = 1;
L1:   if (B == 0) ...     L2:   if (A == 0) ...
```


Assume that the processes are running on different processors, and that locations A and B are originally cached by both processors with the initial value of 0. If writes always take immediate effect and are immediately seen by other processors, it will be impossible for *both* if statements (labeled L1 and L2) to evaluate their conditions as true, since reaching the if statement means that either A or B must have been assigned the value 1. But suppose the write invalidate is delayed, and the processor is allowed to continue during this delay; then it is possible that both P1 and P2 have not seen the invalidations for B and A (respectively) *before* they attempt to read the values. The question is, Should this behavior be allowed, and if so, under what conditions?

The most straightforward model for memory consistency is called *sequential consistency*. Sequential consistency requires that the result of any execution be the same as if the accesses executed by each processor were kept in order and the accesses among different processors were interleaved. This eliminates the possibility of some nonobvious execution in the previous example, because the assignments must be completed before the if statements are initiated. Figure 8.38 illustrates why sequential consistency prohibits an execution where both if statements evaluate to true.

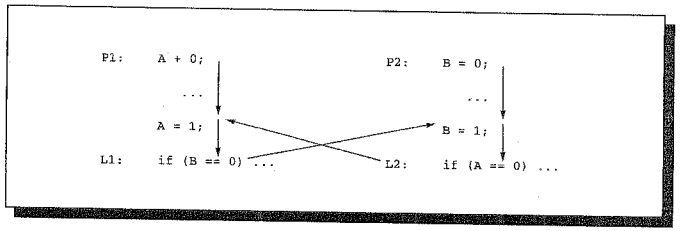

FIGURE 8.38 In sequential consistency, both if statements cannot evaluate to true, since the memory accesses within one process must be kept in program order and the reads of A and B must be interleaved so that one of them completes before the other. To see that this is true, consider the program order shown with black arrows. For both if statements to evaluate to true, the order shown by the two gray arrows must hold, since the reads must appear as if they happen before the writes. For both of these orders to hold and program order to hold, there must be a cycle in the order. The presence of the cycle means that it is impossible to write the accesses down in interleaved order. This means that the execution is not sequentially consistent. You can easily write down all possible orders to help convince yourself.

The simplest way to implement sequential consistency is to require a processor to delay the completion of any memory access until all the invalidations caused by that access are completed. Of course, it is equally simple to delay the

next memory access until the previous one is completed. Remember that memory consistency involves operations among different variables: the two accesses that must be ordered are actually to different memory locations. In our example, we must delay the read of A or B (A==0 or B==0) until the previous write has completed (B=1 or A=1). Under sequential consistency, we cannot, for example, simply place the write in a write buffer and continue with the read. Although sequential consistency presents a simple programming paradigm, it reduces potential performance, especially in a machine with a large number of processors, or long interconnect delays, as we can see in the following Example.

EXAMPLE Suppose we have a processor where a write miss takes 40 cycles to establish ownership, 10 cycles to issue each invalidate after ownership is established, and 50 cycles for an invalidate to complete and be acknowledged. Assuming that four processors share a cache block, how long does a write miss stall the processor if the processor is sequentially consistent? Assume that the invalidates must be explicitly acknowledged before the directory controller knows they are completed. Suppose we could continue executing after obtaining ownership for the write miss without waiting for the invalidates; how long would the write take?

ANSWER When we wait for invalidates, each write takes the sum of the ownership time plus the time to complete the invalidates. Since the invalidates can overlap, we need only worry about the last one, which starts 10 + 10 + 10 + 10 = 40 cycles after ownership is established. Hence the total time is 40 + 40 + 50 = 130 cycles. In comparision, the ownership time is only 40 cycles. With appropriate write-buffer implementations it is even possible to continue before ownership is established. ■

To provide better performance, designers have developed less restrictive memory consistency models that allow for faster hardware. Such models do affect how the programmer sees the machine, so before we discuss these less restrictive models, let's look at what the programmer expects.

The Programmer's View

Although the sequential consistency model has a performance disadvantage, from the viewpoint of the programmer it has the advantage of simplicity. The challenge is to develop a programming model that is simple to explain and yet allows a high performance implementation. One such programming model that allows us to have a more efficient implementation is to assume that programs are *synchronized*. A program is synchronized if all access to shared data is ordered by

synchronization operations. A data reference is ordered by a synchronization operation if, in every possible execution, a write of a variable by one processor and an access (either a read or a write) of that variable by another processor are separated by a pair of synchronization operations, one executed after the write by the writing processor and one executed before the access by the second processor. Cases where variables may be updated without ordering by synchronization are called *data races*, because the execution outcome depends on the relative speed of the processors, and like races in hardware design, the outcome is unpredictable. This leads to another name for synchronized programs: *data-race-free*.

As a simple example, consider a variable being read and updated by two different processors. Each processor surrounds the read and update with a lock and an unlock, both to ensure mutual exclusion for the update and to ensure that the read is consistent. Clearly, every write is now separated from a read by the other processor by a pair of synchronization operations: one unlock (after the write) and one lock (before the read). Of course, if two processors are writing a variable with no intervening reads, then the writes must also be separated by synchronization operations.

We call the synchronization operation corresponding to the unlock a *release*, because it releases a potentially blocked processor, and the synchronization operation corresponding to a lock an *acquire*, because it acquires the right to read the variable. We use the terms acquire and release because they apply to a wide set of synchronization structures, not just locks and unlocks. The next Example shows where the acquires and releases are in several synchronization primitives taken from the previous section.

EXAMPLE Show which operations are acquires and releases in the lock implementation on page 699 and the barrier implementation in Figure 8.34 on page 701.

ANSWER Here is the lock code with the acquire operation shown in bold:

```
lockit: ll    R2,0(R1)    ;load linked
        bnez  R2,lockit   ;not available-spin
        addi  R2,R2,1     ;get locked value
        sc    R2,0(R1)    ;store
        beqz  R2,lockit   ;branch if store fails
```

The release operation for this lock is simply a store operation.

Here is the code for the barrier operation with the acquires shown in bold and the releases in italics (there are two acquires and two releases in the barrier):

```
lock (counterlock);/* ensure update atomic */
if (count==0) release=0;/*first=>reset release */
count++;/* count arrivals */
unlock(counterlock);/* release lock */
if (count==total) { /* all arrived */
      count=0;/* reset counter */
      release=1;/* release processes */
}
else{ /* more to come */

      spin (release=1);/* wait for arrivals */
}
```

■

We can now define when a program is synchronized using acquires and releases. A program is *synchronized* if every execution sequence containing a write by a processor and a subsequent access of the same data by another processor contains the following sequence of events:

```
write (x)
...
release (s)
...
acquire (s)
...
access(x)
```

It is easy to see that if all such execution sequences look like this, the program is synchronized in the sense that accesses to shared data are always ordered by synchronization and that data races are impossible.

It is a broadly accepted observation that most programs are synchronized. This observation is true primarily because if the accesses were unsynchronized, the behavior of the program would be quite difficult to determine because the speed of execution would determine which processor won a data race and thus affect the results of the program. Even with sequential consistency, reasoning about such programs is very difficult. Programmers could attempt to guarantee ordering by constructing their own synchronization mechanisms, but this is extremely tricky, can lead to buggy programs, and may not be supported architecturally, meaning that they may not work in future generations of the machine. Instead, almost all programmers will choose to use synchronization libraries that are correct and optimized for the machine and the type of synchronization. A standard

synchronization library can classify the operations used for synchronization in the library as releases or acquires, or sometimes as both, as, for example, in the case of a barrier.

The major use of unsynchronized accesses is in programs that want to avoid synchronization cost and are willing to accept an inconsistent view of memory. For example, in a stochastic program we may be willing to have a read return an old value of a data item, because the program will still converge on the correct answer. In such cases we still require the system to behave in a coherent fashion, but we do not need to rely on a well-defined consistency model.

Beyond the synchronization operations, we also need to define the ordering of memory operations. There are two types of restrictions on memory orders: *write fences* and *read fences*. Fences are fixed points in a computation that ensure that no read or write is moved across the fence. For example, a write fence executed by processor P ensures that

- all writes by P that occur before P executed the write fence operation have completed, and
- no writes that occur after the fence in P are initiated before the fence.

In sequential consistency, all reads are read fences and all writes are write fences. This limits the ability of the hardware to optimize accesses, since order must be strictly maintained.

From a performance viewpoint, the processor would like to execute reads as early as possible and complete writes as late as possible. Fences act as boundaries, forcing the processor to order reads and writes with respect to the fence. Although a write fence is a two-way blockade, it is most often used to ensure that writes have completed, since the processor wants to delay write completion. Thus the typical effect of a write fence is to cause the program execution to stall until all outstanding writes have completed, including the delivery of any associated invalidations.

A read fence is also a two-way blockade, marking the earliest or latest point that a read may be executed. Most often a read fence is used to mark the earliest point that a read may be executed.

A *memory fence* is an operation that acts as both a read and a write fence. Memory fences enforce ordering among the accesses of different processes. Within a single process we require that program order always be preserved, so reads and writes of the same location cannot be interchanged.

The weaker consistency models discussed in the next section provide the potential for hiding read and write latency by defining fewer read and write fences. In particular, synchronization accesses act as the fences rather than ordinary accesses.

Relaxed Models for Memory Consistency

Since most programs are synchronized and since a sequential consistency model imposes major inefficiencies, we would like to define a more relaxed model that allows higher performance implementations and still preserves a simple programming model for synchronized programs. In fact, there are a number of relaxed models that all maintain the property that the execution semantics of a synchronized program is the same under the model as it would be under a sequential consistency model. The relaxed models vary in how tightly they constrain the set of possible execution sequences, and thus in how many constraints they impose on the implementation.

To understand the variations among the relaxed models and the possible implications for an implementation, it is simplest if we define the models in terms of what orderings among reads and writes *performed by a single processor* are preserved by each model. There are four such orderings:

1. R → R: a read followed by a read.
2. R → W: a read followed by a write, which is always preserved if the operations are to the same address, since this is an antidependence.
3. W → W: a write followed by a write, which is always preserved if they are to the same address, since this is an output dependence.
4. W → R: a write followed by a read, which is always preserved if they are to the same address, since this is a true dependence.

If there is a dependence between the read and the write, then uniprocessor program semantics demand that the operations be ordered. If there is no dependence, the memory consistency model determines what orders must be preserved. A sequential consistency model requires that all four orderings be preserved and is thus equivalent to assuming a single centralized memory module that serializes all processor operations, or to assuming that all reads and writes are memory barriers.

When an order is relaxed, it simply means that we allow an operation executed later by the processor to complete first. For example, relaxing the ordering W→R means that we allow a read that is later than a write to complete before the write has completed. Remember that a write does not complete until all its invalidations complete, so letting the read occur after the write miss has been handled but before the invalidations are done does not preserve the ordering.

A consistency model does not, in reality, restrict the ordering of events. Instead, it says what possible orderings can be *observed*. For example, in sequential consistency, the system must appear to preserve the four orderings just described, although in practice it can allow reordering. This subtlety allows implementations to use tricks that reorder events without allowing the reordering to be

observed. Under sequential consistency an implementation can, for example, allow a processor, P, to initiate another write before an earlier write is completed, as long as P does not allow the value of the later write to be seen before the earlier write has completed. For simplicity, we discuss what orderings must be preserved, with the understanding that the implementation has the flexibility to preserve fewer orderings if only the preserved orderings are visible.

The consistency model must also define the orderings imposed between synchronization variable accesses, which act as fences, and all other accesses. When a machine implements sequential consistency, all reads and writes, including synchronization accesses, are fences and are thus kept in order. For weaker models, we need to specify the ordering restrictions imposed by synchronization accesses, as well as the ordering restrictions involving ordinary variables. The simplest ordering restriction is that every synchronization access is a memory fence. If we let S stand for a synchronization variable access, we could also write this with the ordering notation just shown as S→W, S→R, W→S, and R→S. Remember that a synchronization access is also an R or a W and its ordering is affected by other synchronization accesses, which means there is an implied ordering S→S.

The first model we examine relaxes the ordering between a write and a read (to a different address), eliminating the order W→R; this model was first used in the IBM 370 architecture. Such models allow the buffering of writes with bypassing by reads, which occurs whenever the processor allows a read to proceed before it guarantees that an earlier write by that processor has been seen by all the other processors. This model allows a machine to hide some of the latency of a write operation. Furthermore, by relaxing only this one ordering, many applications, even those that are unsynchronized, operate correctly, although a synchronization operation is necessary to ensure that a write completes before a read is done. If a synchronization operation is executed before the read (i.e. a pattern W...S...R), then the orderings W→S and S→R ensure that the write completes before the read. *Processor consistency* and *total store ordering* (TSO) have been used as names for this model, and many machines have implicitly selected this model. This model is equivalent to making the writes be write fences. We summarize all the models, showing the orderings imposed, in Figure 8.39 and show an example in Figure 8.40.

If we also allow nonconflicting writes to potentially complete out of order, by relaxing the W →W ordering, we arrive at a model that has been called *partial store ordering* (PSO). From an implementation viewpoint, it allows pipelining or overlapping of write operations, rather than forcing one operation to complete before another. A write operation need only cause a stall when a synchronization operation, which causes a write fence, is encountered.

The third major class of relaxed models eliminates the R → R and R → W orderings, in addition to the other two orders. This model, which is called *weak ordering*, does not preserve ordering among references, except for the following:

- A read or write is completed before any synchronization operation executed in program order by the processor after the read or write.
- A synchronization operation is always completed before any reads or writes that occur in program order after the operation.

As Figure 8.39 shows, the only orderings imposed in weak order are those created by synchronization operations. Although we have eliminated the R → R and R → W orderings, the processor can only take advantage of this if it has nonblocking reads. Otherwise the processor implicitly implements these two orders, since no further instructions can be executed until the R is completed. Even with nonblocking reads, the processor may be limited in the advantage it obtains from relaxing the read orderings, since the primary advantage occurs when the R causes a cache miss and the processor is unlikely to be able to keep busy for the tens to hundreds of cycles that handling the cache miss may take. In general, the major advantage of all weaker consistency models comes in hiding write latencies rather than read latencies.

A more relaxed model can be obtained by extending weak ordering. This model, called *release consistency,* distinguishes between synchronization operations that are used to *acquire* access to a shared variable (denoted S_A) and those that *release* an object to allow another processor to acquire access (denoted S_R). Release consistency is based on the observation that in synchronized programs an acquire operation must precede a use of shared data, and a release operation must follow any updates to shared data and also precede the time of the next acquire. This allows us to slightly relax the ordering by observing that a read or write that precedes an acquire need not complete before the acquire, and also that a read or write that follows a release need not wait for the release. Thus the orderings that are preserved involve only S_A and S_R, as shown in Figure 8.39; as the example in Figure 8.40 shows, this model imposes the fewest orders of the five models.

To compare release consistency to weak ordering, consider what orderings would be needed for weak ordering, if we decompose each S in the orderings to S_A and S_R. This would lead to eight orderings involving synchronization accesses and ordinary accesses plus four orderings involving only synchronization accesses. With such a description, we can see that four of the orderings required under weak ordering are *not* imposed under release consistency: W → S_A, R→ S_A, S_R → R, and S_R → W.

Release consistency provides one of the least restrictive models that is easily checkable, and ensures that synchronized programs will see a sequentially consistent execution. While most synchronization operations are either an acquire or a release (an acquire normally reads a synchronization variable and atomically updates it, while a release usually just writes it), some operations, such as a barrier, act as both an acquire and a release and cause the ordering to be equivalent to weak ordering.

Model	Used in	Ordinary orderings	Synchronization orderings
Sequential consistency	Most machines as an optional mode	R→R, R→W, W→R, W→W	S→W, S→R, R→S, W→S, S→S
Total store order or processor consistency	IBMS/370, DEC VAX, SPARC	R→R, R→W, W→W	S→W, S→R, R→S, W→S, S→S
Partial store order	SPARC	R→R, R→W	S→W, S→R, R→S, W→S, S→S
Weak ordering	PowerPC		S→W, S→R, R→S, W→S, S→S
Release consistency	Alpha, MIPS		S_A→W, S_A→R, R→S_A, W→S_R, S_A→S_A, S_A→S_R, S_R→S_A, S_R→S_R

FIGURE 8.39 The orderings imposed by various consistency models are shown for both ordinary accesses and synchronization accesses. The models grow from most restrictive (sequential consistency) to least restrictive (release consistency), allowing increased flexibility in the implementation. The weaker models rely on fences created by synchronization operations, as opposed to an implicit fence at every memory operation. S_A and S_R stand for acquire and release operations, respectively, and are needed to define release consistency. If we used the notation SA and SR for each S consistently, each ordering with one S would become two orderings (e.g., S→W becomes S_A→W, S_R→W), and each S→S would become the four orderings shown in the last line of the bottom-right table entry.

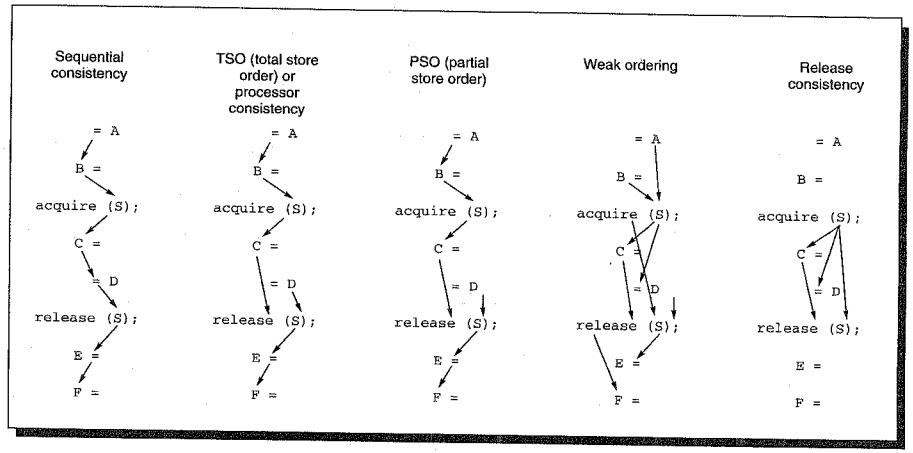

FIGURE 8.40 These examples of the five consistency models discussed in this section show the reduction in the number of orders imposed as the models become more relaxed. Only the minimum orders are shown with arrows. Orders implied by transitivity, such as the write of C before the release of S in the sequential consistency model, are not shown.

It is also possible to consider even weaker orderings. For example, in release consistency we do not associate memory locations with particular synchronization variables. If we required that the same synchronization variable, V, always be acquired before accessing a particular memory location, M, for example, we could relax the ordering of access to M and acquires and releases of all other synchronization variables other than V. The orderings discussed so far are relatively straightforward to implement. Weaker orderings, such as the previous example, are harder to implement, and it is unclear whether the advantages of weaker orderings would justify their implementation.

Implementation of Relaxed Models

Relaxed models of consistency can usually be implemented with little additional hardware. Most of the complexity lies in implementing memory or interconnect systems that can take advantage of a relaxed model. For example, if the memory or interconnect does not allow multiple outstanding accesses from a processor, then the benefits of the more ambitious relaxed models will be small. Fortunately, most of the benefit can be obtained by having a small number of outstanding writes and one outstanding read.

In this section we describe straightforward implementations of processor consistency and release consistency. Our directory protocols already satisfy the constraints of sequential consistency, since the processor stalls until an operation is complete and the directory first invalidates all sharers before responding to a write miss.

Processor consistency (or TSO) is typically implemented by allowing read misses to bypass pending writes. A write buffer that can support a check to determine whether any pending write in the buffer is to the same address as a read miss, together with a memory and interconnection system that can support two outstanding references per node, is sufficient to implement this scheme. Qualitatively, the advantage of processor consistency over sequential consistency is that it allows the latency of write misses to be hidden.

Release consistency allows additional write latency to be hidden, and if the processor supports nonblocking reads, allows the read latency to be hidden also. To allow write latency to be hidden as much as possible, the processor must allow multiple outstanding writes and allow read misses to bypass outstanding writes. To maximize performance, writes should complete and clear the write buffer as early as possible, which allows any dependent reads to go forward. Supporting early completion of writes requires allowing a write to complete as soon as data are available and before all pending invalidations are completed (since our consistency model allows this). To implement this scheme, either the directory or the original requester can keep track of the invalidation count for each outstanding write. After each invalidation is acknowledged, the pending invalidation count for that write is decreased. We must ensure that all pending invalidates to all outstanding writes complete before we allow a release to complete, so we simply check the pending invalidation counts on any outstanding write when a release is executed. The release is held up until all such invalidations for all outstanding writes complete. In practice, we limit the number of outstanding writes, so that it is easy to track the writes and pending invalidates.

To hide read latency we must have a machine that has nonblocking reads; otherwise, when the processor blocks, little progress will be made. If reads are nonblocking we can simply allow them to execute, knowing that the data dependences will preserve correct execution. It is unlikely, however, that the addition of nonblocking reads to a relaxed consistency model will substantially enhance performance. The limited gain occurs because the read miss times in a multiprocessor are likely to be large and the processor can provide only limited ability to hide this latency. For example, if the reads are nonblocking but the processor executes in order, then the processor will almost certainly block for the read after a few cycles. If the processor supports nonblocking reads and out-of-order execution, it will block as soon as any of its buffers, such as the reorder buffer or reservation stations, are full. (See Chapter 4 for a discussion of full buffer stalls in dynamically scheduled machines.) This is likely to happen in at most tens of cycles, while a miss may cost a hundred cycles. Thus, although the gain may be limited, there is a positive synergy between nonblocking loads and relaxed consistency models.

Performance of Relaxed Models

The performance potential of a more relaxed consistency model depends on both the capabilities of the machine and the particular application. To examine the performance of a memory consistency model, we must first define a hardware environment. The hardware configurations we consider have the following properties:

- The pipeline issues one instruction per clock cycle and is either statically or dynamically scheduled. All functional unit latencies are one cycle.
- Cache misses take 50 clock cycles.
- The CPU includes a write buffer of depth 16.
- The caches are 64 KB and have 16-byte lines.

To give a flavor of the tradeoffs and performance potential with different hardware capabilities, we consider four hardware models:

1. *SSBR (statically scheduled with blocking reads)*—The processor is statically scheduled and reads that miss in the cache immediately block.

2. *SS (statically scheduled)*—The processor is statically scheduled but reads do not cause the processor to block until the result is used.
3. *DS16 (dynamically scheduled with a 16-entry reorder buffer)*—The processor is dynamically scheduled and has a reorder buffer that allows up to 16 outstanding instructions of any type, including 16 memory access instructions.
4. *DS64 (dynamically scheduled with a 64-entry reorder buffer)*—The processor is dynamically scheduled and has a reorder buffer that allows up to 64 outstanding instructions of any type. This reorder buffer is potentially large enough to hide the total cache miss latency of 50 cycles.

Figure 8.41 shows the relative performance for two of the parallel program benchmarks, LU and Ocean, for these four hardware models and for two different consistency models: total store order (TSO) and release consistency. The performance is shown relative to the performance under a straightforward implementation of sequential consistency. Relaxed models offer a much larger performance gain on Ocean than on LU. This is simply because Ocean has a much higher miss rate and has a significant fraction of write misses. In interpreting the data in Figure 8.41, remember that the caches are fairly small. Most designers would increase the cache size before including nonblocking reads or even beginning to think about dynamic scheduling. This would dramatically reduce the miss rate and the possible advantage from the relaxed model at least for these applications.

Final Remarks on Consistency Models

At the present time, most machines being built support some sort of weak consistency model, varying from processor consistency to release consistency, and almost all also support sequential consistency as an option. Since synchronization is highly machine specific and error prone, the expectation is that most programmers will use standard synchronization libraries and will write synchronized programs, making the choice of a weak consistency model invisible to the programmer and yielding higher performance. Yet to be developed are ideas of how to deal with nondeterministic programs that do not rely on getting the latest values. One possibility is that programmers will not need to rely at all on the timing of updates to variables in such programs; the other possibility is that machine-specific models of update behavior will be needed and used. As remote access latencies continue to increase relative to processor performance, and as features that increase the potential advantage of relaxed models, such as nonblocking caches, are included in more processors, the importance of choosing a consistency model that delivers both a convenient programming model and high performance will increase.